VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Optické rozpoznávání rostlin pomocí neuronové sítě

Optical Recognition of Plants Using Neural Network

Bc. Petr Dvořáček

Diplomová práce

Vedoucí práce: doc. Ing. Petr Bilík, Ph.D.

Ostrava, 2021

Abstrakt

S rostoucím tlakem na snížení používání chemikálií v zemědělství a zároveň s rostoucí cenou lidské práce je patrná tendence automatizovat postup odstraňování plevele v zemědělství. Tato práce se zabývá vývojem algoritmu, který umožní detekovat zemědělskou plodinu (rostlinu cukrové řepy) v tzv. *nebezpečné* oblasti pomocí kamerového snímku této oblasti. V této nebezpečné oblasti může probíhat automatické mechanické odstraňování plevele pouze v případě, že v ní není detekována zemědělská plodina. Algoritmus pro detekci plodiny je založen na neuronové síti architektury MobileNetV2 [1]. Tato neuronová síť byla natrénována na vytvořeném datasetu čítajícím 73 600 snímků, přičemž bylo dosaženo celkové přesnosti rozpoznání plodiny v 95 % případů. Předpokladem pro zvýšení přesnosti je zvýšení množství snímků v datasetu. Nasazením na jednodeskový počítač NVIDIA Jetson Nano bylo dosaženo rychlosti zpracování obrazu 40 snímků za sekundu.

Klíčová slova

Automatizace v zemědělství, strojové vidění, neuronové sítě, NVIDIA Jetson, TensorFlow, TensorRT

Abstract

The increasing cost of manual work together with the increasing pressure from governments on minimizing of the chemical usage in agriculture intensifies the need for automation in the weed removal process. This work aims to present an algorithm capable of detection of an agricultural crop (sugar beet plant) in a so-called *dangerous* zone. The process of the automatic mechanical removal in the dangerous zone can only continue if the crop is not detected in this area. The detection algorithm is based on a neural network of the MobileNetV2 [1] architecture. After the training on an acquired dataset of 73,600 images, the accuracy of detection using this algorithm was 95 %. To increase the accuracy of the detection even more significantly, it would be necessary to have more training images. The algorithm was deployed on a single-board computer NVIDIA Jetson Nano with the image processing pipeline speed of 40 frames per second.

Keywords

Automation in agriculture, machine vision, neural networks, NVIDIA Jetson, TensoFlow, TensorRT

Poděkování

Tímto bych chtěl poděkovat doc. Ing. Petru Bilíkovi, Ph.D. za svědomité vedení práce. Dále bych chtěl poděkovat Martinu Ullmannovi a jeho týmu za možnost konzultací k problematice i za poskytnuté zázemí v podobě počítačového vybavení. V neposlední řadě také děkuji své rodině za dychtivou podporu i své přítelkyni Barboře za trpělivost a jazykovou korekturu této práce.

Obsah

Seznam obrázků

Seznam tabulek

Zkratky

API Application Programming Interface – rozhraní. 24, 25

CNN Convolutional Neural Network – konvolučí neuronová síť. 15

ILSVRC ImageNet Large Scale Visual Recognition Competition – ImageNet velká soutěž v rozpoznávání obrazu. 8, 15, 33

OOP Object-Oriented Programming – objektově orientované programování. 22

PASCAL VOC Pattern Analysis, Statistical Modelling and Computational Learning – analýza vzorů, statistické modelování a strojové učení. 6, 9, 14–17, *Slovník:* PASCAL VOC

SoM System on Module – systém na modulu. 39

TFLOPS Tera Floating Point Operations Per Second – bilión operací s plovoucí čárkou za sekundu. 40

TOPS Tensor Operations Per Second – tenzorové operace za sekundu. 40

Slovník

open-source Označení pro tzv. *otevřený software*, jehož zdrojový kód je *otevřený*. V praxi to znamená, že kdokoliv může implementovat jakoukoli schálenou funkcionalitu. Příkladem open-source softwaru je Linux [2] nebo TensorFlow[3]. 23

PASCAL VOC Projekt, díky kterému vznikly PASCAL VOC datasety (2005 - 2012). Každý dataset obsahuje řádově desítky kategorií a desetitisíce fotek. Na fotkách jsou označeny objekty daných kategorií ohraničujícím boxem. Je používán především pro hodnocení kvality predikce neuronové sítě pro detekci objektu. 6, 9, 15, 17

TensorFlow Knihovna usnadňující implementaci ML algoritmů. 4, 6, 10, 22–24

Top 5 přesnost Pravděpodobnost s jakou bude správná odpověď mezi pěti predikcemi sítě s nejvyšší pravděpodobností. Pokud je správná odpověď *pudl*, ale predikce sítě seřazeny dle pravděpodobnosti sestupně jsou *teriér, akita, mastif, labrador, pudl, ...*, bude tato předpověď považována za správnou s přesností Top 5. 15

Kapitola 1

Úvod

Omezení používání herbicidů (látek pro hubení plevele) a snížení podílu manuální práce představuje v dnešní době jedno z hlavních témat v zemědělství. Motivací pro snížení nutnosti použití pesticidů při pěstování nejen cukrové řepy je hlavně zvýšení udržitelnosti tohoto průmyslového segmentu. Z pohledu zemědělce, kterému jde hlavně o zisk, se nabízí jako hlavní motivace zvýšení kvality a tedy i ceny vypěstovaného produktu – maximalizace zisku na omezené pěstební ploše zemědělce. Navíc bez nutnosti skladovat herbicidy odpadá i nutnost určitých byrokratických úkonů pro nákup a skladování těchto látek, což se může také pozitivně projevit v nákladech na pěstování zemědělských produktů v bio kvalitě.

V případě, že pole není pravidelně ošetřeno postřikem herbicidu, toto pole brzy zaroste plevelem a výnosy se sníží na minimum. Růstu plevele je třeba zamezit, což lze krom použití chemického postřiku udělat i manuálním odstraněním plevele. V praxi to funguje tak, že celé pole je průběžně zbavováno plevele manuálně. Manuální odstraňování plevele je mimořádně náročné jak finančně, tak z pohledu organizace (je třeba získat brigádníky, někde je ubytovat), proto by bylo skvělé mít k dispozici zařízení, které by mohl farmář připojit za traktor a pomocí něj zbavit pole plevele rychleji a efektivněji.

Cílem této práce je vyvinout algoritmus, díky němuž by mohlo vzniknout zařízení pro automatické mechanické odstraňování plevele **mezi rostlinami v řádku**. Vstupem do algoritmu by měl být snímek tzv. **nebezpečné oblasti** – tj. oblast, ve které probíhá odstraňování rostlin. Výstupem algoritmu by měla být binární hodnota, informace o tom, zda v této nebezpečné oblasti je, nebo není zemědělská plodina (v případě této práce se jedná o cukrovou řepu). Tato informace může být dále využita mechanickým systémem, který v případě výskytu plodiny v nebezpečné zóně proces odstraňování rostlin přeruší. Algoritmus by měl být optimalizovaný pro běh na úsporném zařízení.

Jako zemědělská plodina pro tuto práci byla zvolena cukrová řepa. Automatické odstranění plevele mezi rostlinami cukrové řepy je složitější než například odstranění plevele mezi rostlinami salátu. Důvodem je skutečnost, že cukrová řepa se na poli pěstuje od semene, kdežto salát se

pěstuje tak, že se na rostlin kompletně zbavené pole vysadí již předpěstovaná sazenice salátu – je tedy několikanásobně větší než rostliny plevele, tedy mnohem lépe odlišitelná.

V současnosti již na trhu existují plečky umožňující ničení plevele mezi rostlinami zemědělských plodin v jednom řádku, jako například Robovator [4]. Vzhledem k tomu, že se jedná výhradně o komerční produkty, není jasné na jakém algoritmu je detekce v tomto zařízení postavena či jaké přesnosti je při rozpoznávání plodin dosaženo. Z konzultace se zemědělci vyplynulo, že v tomto segmentu trhu je prostor pro alternativu k produktu Robovator.

Kapitola 2

Seznámení s problematikou

V této práci je řešen problém automatické detekce a rozeznání zemědělských plodin od plevele za účelem zautomatizování procesu odstraňování plevele. Úkolem je vyvinout algoritmus schopný rozpoznání zemědělské plodiny na snímku – tedy v určité oblasti. Informace o přítomnosti plevele v určité oblasti může být převzata dalším systémem, který je zodpovědný za samotné odstraňování plevele (software napojený na mechanickou část plecího stroje). Konkrétně se práce zabývá rozpoznáním rostlin cukrové řepy.

K tomuto účelu byly použity moderní metody pro zpracování a klasifikaci obrazu, jinými slovy **strojového vidění**. Konkrétně v této práci byla použita metoda klasifikace snímků pomocí hlubokých konvolučních neuronových sítí. V následujících podkapitolách je prezentována historie strojového vidění s podstatnými milníky v této oblasti i současnost strojového vidění, která zmiňuje hlavně hluboké konvoluční neuronové sítě i aktuální výzvy a problémy v této oblasti. Dále je podrobně rozebrána samotná problematika neuronových sítí i vybrané vrstvy neuronových sítí.

2.1 Historie strojového vidění

Za nejvlivnější práci v historii strojového vidění je považován článek z oboru neurofyziologie od Davida Hubela a Torstena Wiesela nesoucí název **Receptive fields of single neurons in the cat's striate cortex** [5] z roku 1959. V této studii bylo předmětem zkoumání lokalizace excitovaných neuronů v kočičím mozku, konkrétně v oblasti zodpovědné za vidění. Předpokladem bylo, že po ukázání různých obrazců jednomu (kočičímu) oku budou excitovány různé oblasti mozku. Pokusy se zpočátku nejevily úspěšně. Nakonec bylo zjištěno, že určité části mozku byly dostatečně excitovány pohledem na jednoduché obrazce, zejména jako je přímka. Různé oblasti mozku byly také excitovány při posunu přímky různým směrem.

Výsledkem práce [5] je fakt, že řetězec zpracující obraz u kočky začíná jednoduchými strukturami neuronů, které detekují hrany s různým sklonem. Obrazce hran s různým sklonem, případně i v pohybu, excitují různé struktury neuronů.

Další pro rozvoj oboru strojové vidění zásadní práce Davida Marra publikovaná v roce 1982 **Vision: A computational investigation into the human representation and processing of visual information** [6] staví na myšlence Davida Hubela a Torstena Wiesela [5]. Marr s poznáním o vidění živočichů zachází ještě dále. Z práce vyplývá, že vidění má hierarchistickou strukturu, kde jako první jsou v obrazci detekovány **hrany**, **rohy**, **křivky**. Tyto **nízko dimenzionální** informace jsou zpracovány další vrstvou, která extrahuje informace o hloubce obrazu a povrchu který je na obrazci. Teprve v další vrstvě je vytvořen 3D model vnímaný živočichem, tedy i člověkem. Tato práce bohužel nepředstavila žádný konkrétní algoritmus, či matematický model, který by přímo vedl k implementaci fungujícího umělého vizuálního systému.

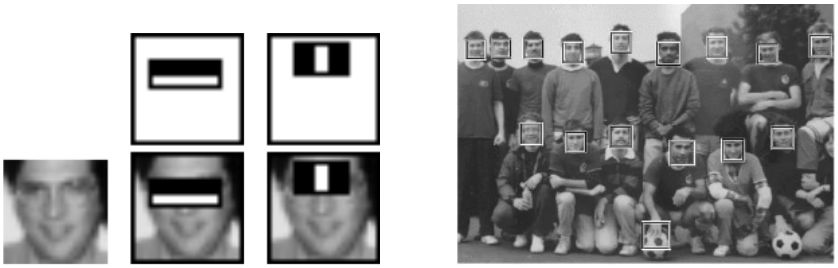
V roce 1979 Japonský informatik Kunihiko Fukushima představil historicky první hlubokou neuronovou síť určenou ke zpracování obrazu, která byla schopna učení na předložených datech (byla samoorganizující se) a zároveň byla díky konvolučním vrstvám translačně invariantní. Síť nese jméno *Necognitron* [7]. Tato práce inspirovala další vývoj v této oblasti.

Do sítě Necognitron v roce 1989 implementoval **Yann LeCun** zpětně propagační algoritmus, což vedlo k zefektivnění procesu učení. Později vytvořil svou vlastní, ve své době průkopnickou architekturu **LeNet-5** [8]. Tato už byla velmi blízká moderním konvolučním neuronovým sítím. Úkolem sítě bylo rozpoznávat směrovací čísla na poštovních zásilkách ve Spojených státech amerických, tzv. ZIP kódů. Jednalo se o komerční produkt. Tato práce také přispěla k vytvoření MNIST datasetu ručně psaných čísel. Tento dataset se dodnes používá k evaluaci kvality předpovědí sítí pro rozpoznání obrazu.

V roce 2001 bylo představeno první real-time řešení pro detekci lidského obličeje [9]. Toto řešení bylo založeno na algoritmu, který pomocí techniky Adaboost vyselektoval tzv. Haarovy příznaky (Haar-like features) ze snímku vhodné pro detekci obličeje (obrázek 2.1 vlevo). Pět let po publikaci bylo řešení implementováno do vybraných digitálních kamer Fujitsu (obrázek 2.1 vpravo). Tento systém je v některých digitálních kamerách používán dodnes.

Roku 2005 byl spuštěn projekt PASCAL VOC (dostupný na [10]) jehož cílem bylo poskytnout standardizovaný dataset pro potřeby srovnání kvality predikce mezi různými architekturami neuronových sítí (případně jiných algoritmů). Výstupem tohoto projektu jsou veřejné datasety s objekty označenými ve standardním formátu pro detekci objektů (značka obsahuje informaci o třídě objektu i o jeho pozici). Nově označený dataset je od roku 2005 zveřejněn každý rok. Spolu se zveřejněním datasetu je vyhlášena soutěž ve které vyhraje autor sítě s nejvyšší přesností. Toto umožnilo objektivní porovnání predikce různých architektur i implementací neuronových sítí. V publikacích prezentujících nové architektury je obvykle právě jeden z parametrů deklarující kvalitu sítě **přesnost na PASCAL VOC**. PASCAL VOC 2006 dataset obsahuje celkem asi 10 000 označených obrázků 10 objektů různých kategorií (kolo, autobus, kočka, auto, kráva, pes, kůň, motorka, člověk, ovce) v různých polohách a různých rozměrech (obrázek 2.2) [11].

Dalším důležitým milníkem ve standardizovaném testování kvality různých algoritmů pro rozpoznávání obrazu, případně pro detekci objektů na obrázku, byl rok 2010, kdy byl spuštěn další projekt



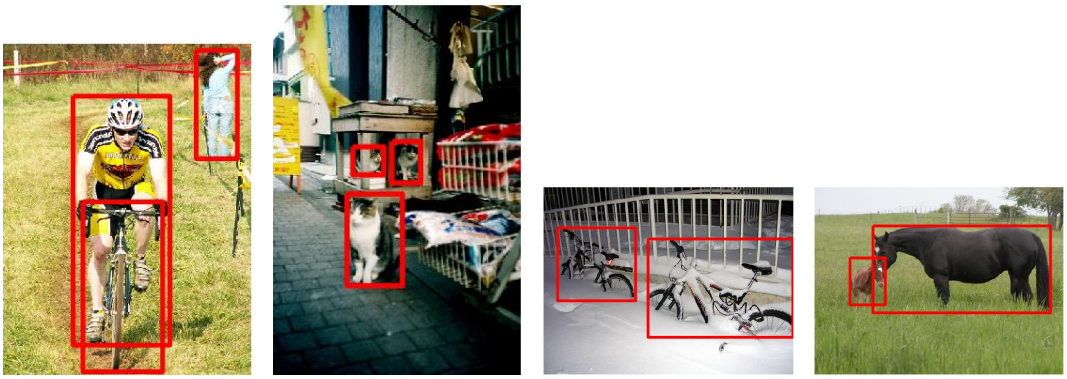
Obrázek 2.1: První dva příznaky vybrané algoritmem Adaboost pro detekci obličeje (vlevo). Výsledek algoritmu pro detekci obličeje (vpravo). Příznaky (horní řádek), obličej z datasetu původní a překrytý příznaky (spodní řádek). První příznak měří rozdíl v intenzitě oblasti očí a oblasti horní líce. První příznak byl vybrán, protože oblast očí je obvykle tmavší než oblast horní líce. Druhý příznak porovnává intenzitu oblasti nosu s intenzitou oblasti nosního můstku [9].

podobný projektu PASCAL VOC – **ImageNet Large Scale Visual Recognition Competition – ImageNet velká soutěž v rozpoznávání obrazu (ILSVRC)** (dostupný na [12]). Dataset, který díky tomuto projektu vznikl, bude dále označován jako ImageNet. ImageNet obsahuje celkem přibližně 14 milionů obrázků roztříděných do jednoho tisíce kategorií. Jeden milion z těchto obrázků je dokonce anotováno pro detekci objektů, tzn. boxem, který objekt ohraničuje (stejně jako anotace datasetu PASCAL VOC 2006 na obrázku 2.2). Výhoda datasetu ImageNet je zřejmá z obrázku 2.3. ImageNet obsahuje mnohem více podtříd jednoho objektu – místo jedné kategorie *pes* (jako to bylo v datasetu Pascal VOC). ImageNet obsahuje jednotlivá psí plemena jako samostatné kategorie. Toto umožňuje vyhodnotit schopnost algoritmu strojového vidění rozlišit velmi podobné objekty.

Přelomovým rokem pro CNN byl rok 2012, kdy soutěž ILSVRC vyhrála architektura **AlexNet** [14] týmu *SuperVision* s obřím náskokem oproti konkurenci (11%, viz. tabulka 2.1). Síť byla založena právě na konvolučních vrstvách, podobně jako již zmíněná architektura LeNet-5 [8]. Od roku 2012 soutěž ILSVRC vyhrávají výhradně CNN a to vždy jen o jednotky, ne-li desetiny procenta. Takový náskok v přesnosti, jakou měla architektura AlexNet zatím nebyl pozorován – prostoru pro zlepšení je mnohem méně, než tomu bylo dříve.

Tabulka 2.1: Tabulka prvních čtyř vítězů soutěže ILSVRC 2012 [15]

Jméno týmu	Top 5 přesnost
SuperVision (AlexNet)	**0,85**
ISI	0,74
OXFORD_VGG	0,73
XRCE/INRIA	0,73



Obrázek 2.2: Příklady obrázků z datasetu PASCAL VOC 2006. Červené boxy jsou vykreslené značky objektů a byly vytvořeny člověkem v procesu zvaném **anotování**. Originální obrázky tyto boxy neobsahují, slouží pouze k účelu učení neuronové sítě (či jiného algoritmu). Dataset PASCAL VOC obsahuje obrázky s různým rozlišením [11].

2.2 Současnost strojového vidění

Za současnost strojového vidění bude v této práci považováno období od roku 2012. Toto časové období, počínající představením architektury AlexNet [14], bylo zvoleno na základě zdrojů podstatných po tuto práci, jež uvádějí AlexNet právě jako klíčový milník [16, 17, 18, 1]. V současnosti jsou na předních příčkách soutěží v klasifikaci obrazu i detekci objektů umístěny výhradně konvoluční neuronové sítě.

Od publikace sítě AlexNet byly vytvořeny další architektury, které postupně dosahovaly vyšší a vyšší přesnosti na ImageNet datasetu. Zvýšení přesnosti bylo dosaženo zejména zvyšováním počtu konvolučních vrstev sítě. Prapůvodní LeNet-5 [8] i AlexNet [14] obsahovaly oba mimo jiné 5 konvolučních vrstev, VGG [16] až 19 a moderní architektura ResNet [19] jich obsahuje až 151. Vývoj moderních velmi hlubokých sítí není tak přímočarý, jak by se na první pohled mohlo zdát. Není možné pouze přidávat další a další konvoluční vrstvy a očekávat lepší výsledky bez vyřešení jistých problémů. Neuronové sítě totiž při trénování upravují své parametry na základě signálu z výstupu sítě (po spočítání **ztrátové funkce**) a tento signál se při zvýšení počtu vrstev nad určitou hranici nedostane dále než několik vrstev od konce sítě, jednoduše se ztratí. Tento jev je nazýván **problém mizejícího gradientu** (vanishing gradient problem) a způsobí, že k učení dochází pouze u výstupu sítě; parametry, konvoluční filtry na začátku sítě, se nemění a přesnost sítě na testovacím datasetu je značně nižší, než na datasetu trénovacím. Za úspěchem moderních neuronových sítí, mezi něž patří i zmiňovaná architektura ResNet, stojí právě způsob řešení tohoto problému.

Pro reálné nasazení modelů neuronových sítí na zařízení mimo cloudová centra, ve kterých jsou dostupné nejvýkonnější grafické karty, například do embedded zařízení či mobilních telefonů, je podstatná nejen přesnost modelu, ale i velikost modelu a jeho výpočetní složitost. Nejmodernější

Obrázek 2.3: Dataset ImageNet obsahuje mnohem více podtříd v porovnání s datasetem PASCAL VOC. Například PASCAL VOC obsahuje jednu třídu *pes*, která zahrnuje fotky mnoha psích plemen; ImageNet obsahuje 120 tříd *pes*, kde každá z této třídy reprezentuje jedno psí plemeno [13].

architektury v tomto ohledu jsou modely z rodiny MobileNet (MobileNet, MobileNetV2, MobileNetV3 [20, 1, 21]). Tato práce je zaměřena hlavně na architekturu MobileNetV2, protože pro výslednou aplikaci je velmi podstatná rychlost vyhodnocení (tedy rychlost **inference**). Architektura MobileNetV2 je sice o něco méně přesná v porovnání s alternativami, na druhou stranu je ale výrazně rychlejší. V tabulce 2.2 jsou pro představu uvedeny moderní architektury neuronových sítí spolu s počtem jejich parametrů v porovnání s přesností na datasetu ImageNet.

Tabulka 2.2: Přehled přesnosti a počtu parametrů vybraných architektur neuronových sítí na datasetu ImageNet.

Architektura	Top 1 přesnost (%)	Top 5 přesnost (%)	Počet parametrů (mil.)
AlexNet	Neuvedeno	85	57
VGG-19	76	93	140
ResNet-152	81	96	58
MobileNetV2	72	91	3

2.3 Hluboká neuronová síť

Umělá hluboká neuronová síť je konstrukce určená ke zpracovávání dat, původně inspirovaná biologickou neuronovou síti (mozkem). Hluboká neuronová síť je označení pro neuronovou síť, která se skládá z **více než 2 vrstev** – dále v této práci nebude rozlišováno mezi neuronovou sítí a hlubokou neuronovou sítí. Vždy bude slovním spojením *neuronová síť* myšlena hluboká neuronová síť. To, čím se neuronová síť odlišuje od ostatních algoritmů, je fakt, že je schopna zobecnění na základě určité, řekněme *zkušenosti*. Tato zkušenost je neuronové síti předána procesem tzv. trénování. Při tomto procesu je neuronové síti předvedeno velké množství dat – tzv. proces **trénování** neuronové sítě. Úkolem neuronové sítě je nalezení vhodných hodnot trénovatelných parametrů, díky kterým poté bude síť schopna přesné predikce. Neuronová síť natrénovaná na určitý problém může být také nazývána modelem. V kontextu této práce je neuronová síť použita pro úkol klasifikace snímku (snímek na kterém je cukrová řepa, snímek na kterém není cukrová řepa). Konvoluční neuronová síť spadá do kategorie hlubokých dopředných neuronových sítí.

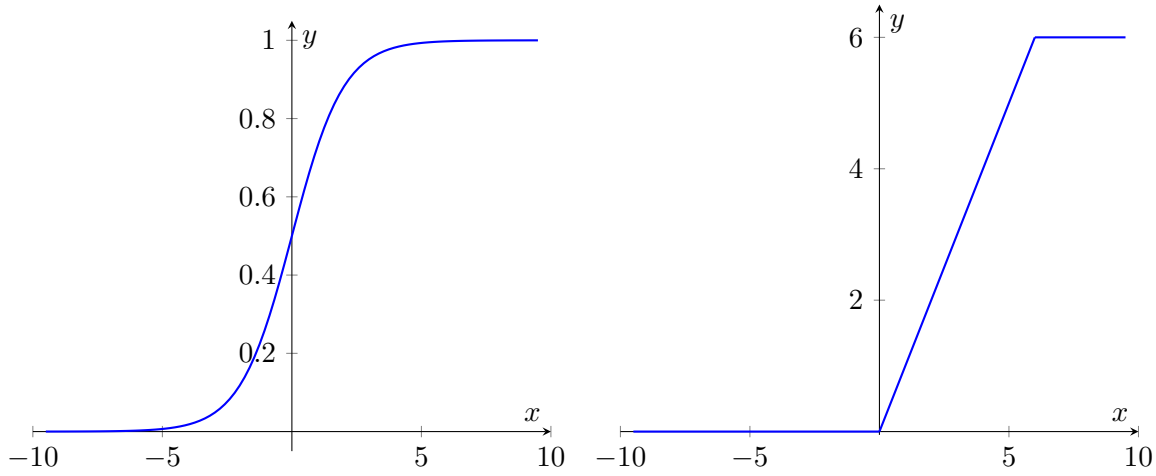
Dopředná neuronová síť (dále zase jen neuronová síť) se skládá z neuronů propojených **váhami**. Tyto neurony jsou uskupeny v jednotlivých vrstvách. Každý neuron v síti je propojen se všemi neurony v předchozí vrstvě (nebo je na něj přiveden vstup neuronové síťě). Výstupem každého neuronu je jedna hodnota, která je zase přivedena na vstup každého neuronu ve vrstvě následující – jakým procesem je tato výstupní hodnota získána, je popsáno v dalším odstavci. Pomocí vah je přiřazena určitá důležitost spojením mezi jednotlivými neurony. Dle nastavení vrstvy je možné každému jejímu neuronu přiřadit i tzv. **bias** (zaujatost). Bias (B) i váhy (W) jsou trénovatelnými parametry. Mezi vstupem (X) a výstupem (Y) jednotlivé vrstvy v síti platí následující vztah

$$Y = \sigma(W \cdot X + B) = \sigma\left(\sum_{i=1}^{n}(w_i \cdot x_i) + b\right) \tag{2.1}$$

přičemž σ je takzvaná aktivační funkce.

V lidském mozku může být neuron ve dvou stavech – excitován (řekněme **logická 1**) a v klidu (**logická 0**). V neuronových sítích se toto striktně diskrétní rozdělení stavů neuronů neosvědčilo a namísto toho je výstup neuronu **spojitý**, navíc je na něj aplikována tzv. **aktivační funkce** σ. Existuje celá řada aktivačních funkcí. Jako jedna z prvních aktivačních funkcí byla v neuronových sítích používána funkce sigmoid, která je definována vztahem $\sigma(x) = \frac{1}{1+e^{-x}}$ a je vyobrazena na obrázku 2.4 Tato funkce se ukázala být zbytečně moc výpočetně náročná, a tak ji postupně nahradily aktivační funkce z rodiny ReLU. Konkrétně v této práci byla použita aktivační funkce ReLU6, pro jejíž výstup platí

$$\sigma(x) = \begin{cases} 0, & \text{if} x \in < -\infty; 0 > \\ x, & \text{if} x \in (0; 6), \\ 6, & \text{if} x \in < 6; \infty > \end{cases} \tag{2.2}$$



Obrázek 2.4: Příklad aktivační funkce sigmoid (vlevo) a ReLU6 (vpravo). Pro tuto práci byla použita aktivační funkce ReLU6.

Sigmoid i ReLU jsou nelineární funkce. Výhoda použití nelineárních funkcí pro aktivaci spočívá v tom, že tyto umožňují neuronové síti snadněji se naučit komplexní reprezentaci dat.

2.3.1 Normalizace vstupu

Normalizace vstupu se provádí za účelem zrychlení procesu trénování sítě a zvýšení její stability. Pro aplikování normalizace je za vrstvu, kterou je cílem normalizovat, přidána vrstva právě pro normalizaci vstupu. Do této normalizační vrstvy vstupuje (X) a její výstup je dán vztahem

$$Z = \frac{X - m}{s} \tag{2.3}$$

přičemž m je střední hodnota vstupu a s je jeho směrodatná odchylka. Tyto hodnoty jsou počítány v kontextu všech vzorků v jednom trénovacím balíčku (tzv. **batch**). Takto normalizovaný tenzor Z je poté vynásoben parametrem g a je k němu přičten tenzor B (reprezentuje *bias*) [22].

Touto metodou je možné zajistit, že do vstupu jednotlivých vrstev vstupují v rámci jednoho balíčku data, která mají ze statistického pohledu nulovou střední hodnotu a směrodatnou odchylku rovnu jedné. Aplikováním normalizace balíčku na vrstvy sítě architektury Inception bylo dosaženo **stejné přesnosti**, jako bez použití této vrstvy v **poloviční čase** a přesnost se zvýšila o 0.5 % [22] – proces trénování je tedy výrazně rychlejší s použitím normalizace vstupu. Tato normalizační vrstva je také použita v architektuře MobileNetV2, na které je založená síť v této práci [1].

2.3.2 Konvoluční vrstva

Neuronové sítě, jejichž úkolem je zpracování obrazu jsou charakteristické využitím tzv. **konvolučních vrstev**. Tato vrstva funguje jako detektor určitých kombinací hran v obrázku. Podle toho, jak

hluboko v síti se daná konvoluční vrstva nachází[1], je vrstva schopna na základě výstupu předchozích konvolučních vrstev detekovat komplexnější tvary i objekty. Obecně platí, že konvoluční vrstva blízko vstupu detekuje texturu, jednotlivé hrany v obrázku, a konvoluční vrstva u výstupu (hluboko) detekuje kombinace těchto objektů – komplexnější objekty jako obličej atd. Pro zpracování obrazu se používá diskrétní konvoluce obrazu (tedy diskrétní konvoluce signálu o dvou proměnných s omezeným definičním oborem). Tuto lze odvodit ze standardní, spojité konvoluce následovně.

Konvoluce mezi dvěma signály $f(x)$ a $g(x)$ je definována jako

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t) \cdot g(x - t)dt \tag{2.4}$$

Z tohoto lze odvodit vzorec pro konvoluci dvou signálů s definičním oborem v celých číslech (diskrétní signály)

$$(f * g)(x) = \sum_{i=-\infty}^{\infty} f(i) \cdot g(x - i) \tag{2.5}$$

Tento matematický aparát nám zjednodušeně řečeno dá informaci o tom, jak moc si jsou signály podobné. Pro naše potřeby je toto dobrý základ, nicméně vzhledem k tomu, že je zpracováván obraz, nikoliv 1D signál, je třeba konvoluci provádět s funkcemi dvou, nikoliv jedné proměnné. Nechť $f(x, y)$ představuje obrázek vstupující do konvoluce (diskrétní 2D pole) definovaný na intervalu $(-\infty; \infty)$ přičemž na intervalu přesahující rozměry obrázku je $f(x, y)$ rovna nule[2] a $g(i, j)$ představuje tzv. jádro konvoluce (konvoluční filtr) definovaný na jiném intervalu $< -k; k >$ (menší interval, typicky 3×3), pak výstupem konvoluce obrázku f konvolučním jádrem g je

$$(f * g)(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x - i, y - j) \cdot g(i, j) \tag{2.6}$$

Toto je matematický základ pro definování konvoluční vrstvy v konvoluční neuronové síti.

Pro jasnost je uveden i konkrétní příklad, pro jednoduchost byly zvoleny malé rozměry jednotlivých matic. Nechť I reprezentuje obrázek o rozměru 3×3 a jednom barevném kanálu. Nad tímto je provedena konvoluce filtrem K o rozměru 2×2, viz. následující rovnice

$$I = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}; K = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}; O = I * K = \begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix} \tag{2.7}$$

Problémem je, že v tomto případě není rozměr vstupu (I) roven rozměru výstupu (O). Toto lze vyřešit tzv. paddingem. Tímto se uměle zvětší velikost vstupního obrázku (I) obvykle přidáním 0

[1] hluboko – blízko k výstupu

[2] Padding, pro umožnění konvoluce i v krajích obrázku

tak, aby po konvoluci měl výsledek (O) rozměry rovné s původními rozměry vstupního obrázku. Prakticky to vypadá následovně

$$I = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}; K = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}; O = I * K = \begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 26 & 10 \\ 18 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix} \tag{2.8}$$

2.3.3 Ztrátová funkce

Definování správné ztrátové funkce je stěžejním úkolem při tvorbě modelu neuronové sítě. Ztrátová funkce definuje úlohu, kterou chceme, aby model úspěšně řešil. V následujícím odstavci bude uveden **ilustrativní** příklad špatně zvolené ztrátové funkce pro jednoduchou (ilustrativní) úlohu. Dále zde bude také popsána ztrátová funkce použita v této práci.

Předpokládejme, že chceme vytvořit algoritmus strojového učení, jehož úkolem by bylo **maximalizovat pocit štěstí lidstva** na celé planetě Zemi. Úloha se zdá na první pohled velmi jednoduchá – zdá se, že je jasně stanovený parametr, podle kterého je možné určit, zda je algoritmus naše požadavky splňuje, nebo je nesplňuje. V takovémto případě chceme maximalizovat **funkci odměny** (opak ztrátové funkce, kterou je třeba minimalizovat). Nicméně v případě, že 1 % lidstva drží 99 % bohatství světa, by algoritmus mohl velmi rychle dojít k závěru, že místo pokusu o zvýšení kvality života v rozvojových zemích, je mnohem jednodušší tuto skupinu obyvatel doslova odříznout, **zahubit** a vytvořit velmi malou skupinu privilegovaných lidí, kterým budou vyhrazeny všechny zdroje a *pravděpodobně* budou šťastnější. Toto je klasický příklad špatně definované ztrátové funkce.

Tato práce se nicméně zabývá **klasifikací obrazu**, nikoliv řešením globálních problémů jako je *pocit štěstí světové populace*. Úkol je, na rozdíl od předchozího ilustrativního příkladu, velmi jasně definován – rozhodni, zda na obrázku řepa je, nebo není. Pro tento účel byla zvolena ztrátová funkce **binární křížové entropie**, která je, na rozdíl od křížové entropie, modifikovaná pro úlohy klasifikace do dvou tříd. Vzorec binární křížové entropie se dá následovně zjednodušit

$$CE = -\sum_{i=1}^{C=2} t_i log(s_i) = -t_1 log(s_1) - (1 - t_1) log(1 - s_1) \tag{2.9}$$

přičemž C je počet tříd, t_i značka a s_i predikce pro danou třídu s indexem i.

Kapitola 3

Rozbor nástrojů

Problematika tvorby a implementace neuronových sítí je velmi obsáhlá, existují již implementované knihovny (zejména v jazyku C++), které se pro účel implementace neuronových sítí standardně používají jak v akademické sféře, tak v průmyslu. Znamená to tedy, že není třeba znovu vynalézat kolo, tedy implementovat neuronovou síť tzv. *na zelené louce*, ale je vhodné těchto knihoven využít. I když jsou samotné knihovny napsané zejména v jazyku C++ a CUDA, obvykle je možné jejich optimalizované funkce volat z tzv. vyššího programovacího jazyka – jazyka **Python**. Konkrétně je řeč o knihovně **TensorFlow**, **Keras** a **TensorRT**. Knihovna TensorFlow představuje *assembler* pro výpočty spojené s neuronovými sítěmi. Pojem assembler byl použit jen jako přirovnání k jazyku, do kterého se kompiluje abstraktnější kód. Tím abstraktnějším kódem je myšlen kód vytvořený knihovnou Keras, která umožňuje přistupovat k neuronovým sítím i jejím vrstvám jako k samostatným objektům – poskytuje vhodnou abstrakci. TensorRT je zase knihovna umožňující rychlostní i paměťové optimalizace již natrénovaného modelu. Pomocí knihovny TensorRT je model zkompilován pro inferenci na jedno konkrétní zařízení. Všechny tyto knihovny jsou psány, jak již bylo uvedeno, v jazyku C++ a CUDA, ale umožňují volání funkcí z jazyka **Python**. Tyto knihovny, spolu s jazykem Python budou blíže popsány v následujících podkapitolách.

3.1 Python

Programovací jazyk Python spadá do kategorie interpretovaných, interaktivních a objektově orientovaných (OOP) jazyků[1]. Ačkoliv paradigmaticky spadá do OOP, podporuje i funkcionální a procedurální paradigmata programování. Dále podporuje organizaci kódu pomocí modulů, výjimky (exceptions), dynamické typování, vysoko úrovňové dynamické datové typy a třídy. Python se vyznačuje čistou a dobře čitelnou syntaxí, zároveň je **rozšiřitelný o knihovny psané v jazyce C a C++** (např. TensorFlow), díky čemuž se často používá jako tzv. frontendový jazyk [23].

[1] Object-Oriented Programming – objektově orientované programování (OOP)

Autorem jazyka je nizozemský programátor **Guido van Rossum**, nicméně od verze jazyka 2.1 je spravován tzv. **Python Software Foundation**, což je nezávislá nezisková organizace držící autorská práva jazyka a spravující jazyk samotný. Jazyk je open-source, je možné jej za určitých podmínek zdarma používat, měnit i prodávat pozměněný, či nepozměněný a samozřejmě i prodávat software v něm vyvinutý [23].

V této práci byl použit jazyk ve verzi 3.6, z důvodu plné kompatibility s operačním systémem použitým pro praktickou část této práce (Mint 19.1 tessa). Pro reprodukci výsledků této práce je doporučeno použít právě tuto verzi jazyka.

3.2 TensorFlow

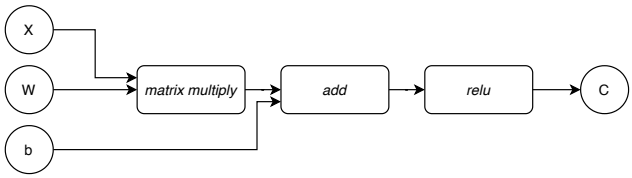
TensorFlow je open-source rozhraní pro implementaci algoritmů s podporou GPU značky NVIDIA. Algoritmus implementovaný pomocí TensorFlow může být nasazen téměř beze změny na široké spektrum hardwarových platforem, od mobilních zařízení přes stolní počítače vybavené jednou nebo více GPU až po klastry vybavené stovkami výpočetních jednotek s tisíci kartami GPU. Implementace algoritmu s použitím knihovny TensorFlow je obecně jednodušší, rychlejší a flexibilnější v porovnáním s implementací přímo v jazyku C++ a CUDA. Obecně se také dá říci, že kód napsaný s využitím knihovny TensorFlow lehce škálovatelný [3].

Jádro knihovny TensorFlow je napsáno v jazyku C++ a CUDA (kvůli podpoře GPU značky NVIDIA). Knihovnu lze využít i v jazycích Python, Java, C, Go přes tzv. *předkompilované* funkce a metody. V této práci byl jako hlavní jazyk zvolen **Python**.

Výpočty TensorFlow jsou vyjádřeny jako tzv. **stavové grafy toku dat**. Je snadné vyjádřit různé druhy paralelismu prostřednictvím replikace a paralelního provedení grafu toku dat základního modelu v TensorFlow [3].

Přesněji řečeno je výpočet v TensorFlow popsán směrovaným grafem, který se skládá ze sady uzlů. Výpočtový graf lze sestavit pomocí jednoho z podporovaných frontendových jazyků, (viz. příklad na obrázku 3.1). Každý uzel má nula nebo více vstupů a nula nebo více výstupů a představuje instanci operace v TensorFlow. Hodnoty, které putují po hranách v grafu (od výstupů po vstupy), jsou tenzory, tedy pole libovolné dimenze. Datové typy tenzorů jsou definovány při konstrukci grafu. V grafu mohou také existovat speciální hrany, které se nazývají kontrolní závislosti, nedochází v nich k toku dat, ale definují závislost operace (zdrojového uzlu) na výsledku předchozí operace (či více operací). Takto je v TensorFlow dosaženo souběžného charakteru algoritmu, který je předpokladem pro paralelní běh algoritmu [3].

Každá operace v TensorFlow je unikátně pojmenovaná a reprezentuje abstraktní výpočet (například *matrix multiply* pro maticové násobení, nebo *add* pro součet). Operace může mít atributy a všechny atributy musí být definovány nebo odvozeny právě v době konstrukce grafu, aby se vytvořila instance uzlu k provedení operace. Atributy umožňují zejména polymorfismus operace – sečtení dvou tenzorů typu float a typu int32 se uživateli knihovny jeví stejně. Jádrem (kernel) je



Obrázek 3.1: Výpočetní graf vygenerovaný knihovnou TensorFlow dle kódu ve výpisu 3.1.

nazývána konkrétní implementace operace, kterou lze spustit na konkrétním typu zařízení (CPU, GPU). Takto TensorFlow abstrahuje hardware, na kterém je výpočet spuštěn. Jádro lze v kontextu TensorFlow považovat za „assemblerový“ jazyk.[2]

```
import tensorflow as tf
W = tf.Variable(tf.random.uniform([64, 64], -1, 1), name="weights")
b = tf.Variable(tf.random.uniform([64, 64], -1, 1), name="biases")
X = tf.Variable(tf.random.uniform([64, 64], -1, 1), name="input", dtype="float32")
C = tf.nn.relu(tf.matmul(W, X) + b)
```

Výpis 3.1: Python kód využívající knihovnu TensorFlow pro jednoduchý výpočet ($C = W \cdot X + b$).

3.3 Keras

Keras je nástroj, který poskytuje další úroveň abstrakce nad výpočty (nejen) v TensorFlow. Běžně se používá při vývoji modelů strojového učení, zejména neuronových sítí. Od verze TensorFlow 2.0 je Keras její nativní součástí.

Keras zapouzdřuje logiku za operacemi v neuronových sítích (spojení vrstev, konvoluce, operace pooling) a poskytuje API pro modelování neuronových sítí, kde základní stavební blok představuje vrstva, nikoliv operace. Výsledný výpočet se poté provádí v TensorFlow.

Příklad definování modelu neuronové sítě pomocí rozhraní Keras API v TensorFlow je ve výpisu 3.2. Vstupem modelu je tříkanálový (RGB) obraz o velikosti 224x224 a výstupem je míra příslušnosti objektu na obrázku do 2 tříd. Tento model byl použit pouze pro demonstraci rozhraní keras API, nebyl použit ve výsledném algoritmu pro detekci vad ve dřevě.

```
neural_network_model = keras.Sequential(
   [
   keras.Input(shape=(224, 224, 3)),
   layers.Conv2D(32, 5, strides=2, activation="relu"),
   layers.Conv2D(32, 3, activation="relu"),
   layers.Conv2D(32, 3, activation="relu"),
   layers.GlobalAveragePooling2D(),
```

[2] pouze přirovnání, jádro TensorFlow je napsáno hlavně v C++ a CUDA, nikoli přímo v assembleru.

```
    layers.Dense(2)
    ]
)
```

Výpis 3.2: Příklad vytvoření jednoduché neuronové sítě s použitím Keras Application Programming Interface – rozhraní (API). Vstupem do sítě je v tomto případě obrázek o rozměrech $224 \times 224 \times 3$ (RGB), výstupem jsou dvě hodnoty míry příslušnosti objektu na obrázku do jedné ze dvou tříd.

3.4 TensorRT

Knihovna TensorRT umožňuje vytvořený, natrénovaný model neuronové sítě optimalizovat pro inferenci. Znamená to, že po optimalizaci již není možné model dále trénovat (tedy upravovat jeho parametry), ale je možné pouze získat jeho predikci v procesu zvaném **inference**. Knihovna dále umožňuje také změnit přesnost datových typů, které jsou použity ve výpočtu z defaultního datového typu **float 32** na datový typ s nižší přesností **float 16**, nebo i **integer 8**. Změnou přesnosti datových typů je možné model zrychlit na úkor kvality predikce. V této práci snížení přesnosti datových typů nebylo využito, protože model dosahuje dostatečné rychlosti už při použití plného datového typu **float 32**. Naopak zde bylo využito celkové optimalizace modelu. V této práci byla použita lehce modifikovaná verze knihovny, a sice verze integrovaná přímo do knihovny TensorFlow.

Samotná kompilace modelu musí být prováděna na zařízení, na které bude nakonec model nasazený – zkompilovaný model je nepřenosný mezi zařízeními různého typu (s rozdílným CPU a GPU). Příklad kódu pro kompilaci modelu je uveden ve výpisu 3.3. Parametr **max_workspace_sizes** představuje maximální hodnotu paměti GPU[3], kterou může zkompilovaný model při vyhodnocení využít. Dále parametr **minimum_segment_size** definuje minimální počet po sobě jdoucích operátorů které mohou být optimalizovány. Pokud je parametr příliš malý, může dojít k tomu, že model po optimalizaci bude pomalejší než bez ní, a to proto, že přepnutí režimu výpočtu z nativního (TensorFlow) do TensorRT se vykonává příliš často a jedná se o časově náročnou operaci. Zase na druhou stranu pokud bude parametr příliš vysoký, může dojít k tomu, že síť nebude možné optimalizovat vůbec.

Knihovna TensorRT provádí optimalizaci pouze těch operací v modelu, které sama podporuje. V případě architektury (resp. natrénovaného modelu) MobileNetV2 jsou všechny operace podporovány. Seznam všech podporovaných operací lze najít v referenci [24].

[3]Na zařízení z rodinyJetson je RAM i paměť GPU sdílená

```
from tensorflow.python.compiler.tensorrt import trt_convert as trt
params = trt.DEFAULT_TRT_CONVERSION_PARAMS._replace(
    max_workspace_size_bytes=(3.6*10**12),
    precision_mode='fp32',
    minimum_segment_size=3)
converter = trt.TrtGraphConverterV2(
    input_saved_model_dir=model_path,
    input_saved_model_tags='serve',
    conversion_params=params)
converter.convert()
converter.save(output_saved_model_dir=output_path)
```

Výpis 3.3: Příklad kompilace natrénovaného modelu s využitím knihovny TensorRT.

Kapitola 4

Tvorba datasetu

Pestrý a reprezentativní dataset je stěžejním předpokladem pro kvalitně natrénovanou neuronovou síť, která poskytuje dobré výsledky nejen v laboratorních podmínkách, ale i v po nasazení v praxi. Kvalitně natrénovaná síť je taková, která je schopná zobecnění, což znamená, že její přesnost bude stejně vysoká při různých podmínkách (například při různých světelných podmínkách). Obecně platí, že čím více různých snímků je použito pro trénování sítě, tím stabilnější a spolehlivější je její predikce.

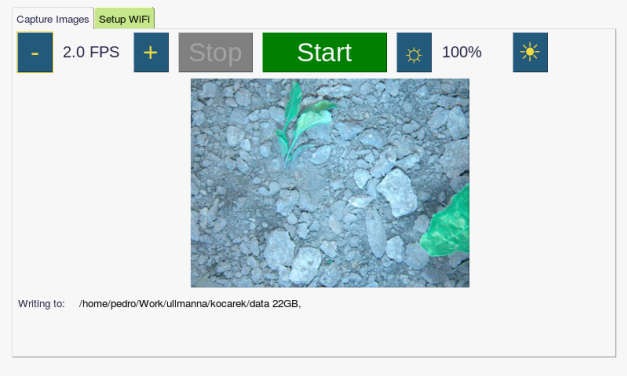
V této sekci bude popsána aplikace, která byla vytvořena za účelem sběru snímků (**proces tvorby datasetu**) přímo v exteriéru na poli. Tato aplikace nese pracovní název **kočárek**. V následujících podkapitolách bude popsána jednak aplikace kočárek a jednak způsob, jakým byly nasbírané snímky (dataset) upraveny, aby mohly být použity pro trénování neuronové sítě.

4.1 Aplikace kočárek

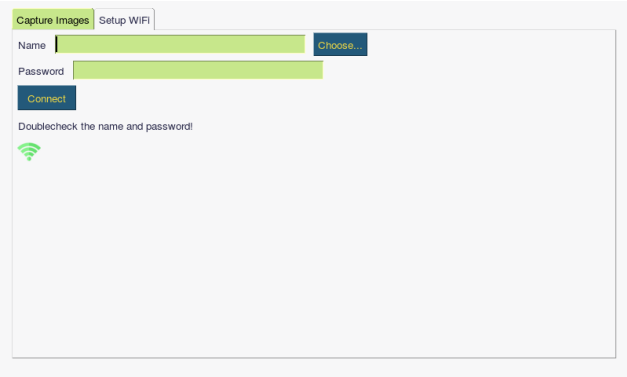
Pro zajištění sběru snímků plodiny (rostliny cukrové řepy) byla v rámci diplomové práce vyvinuta aplikace s pracovním názvem *kočárek*. Grafické rozhraní aplikace je zobrazeno na obrázku. 4.1. Nasbírané a označené snímky jsou určeny k trénování neuronové sítě samotné a také k testování kvality její predikce.

Celá aplikace byla naprogramována v asynchronní architektuře v programovacím jazyku Python s využitím grafických knihoven **PySimpleGUI** a **Tkinter** a asynchronní knihovny asyncio. Asynchronní (tedy neblokující) architektura byla zvolena zejména kvůli intenzivní práci se soubory (čtení dat z kamery, zápis snímků na disk).

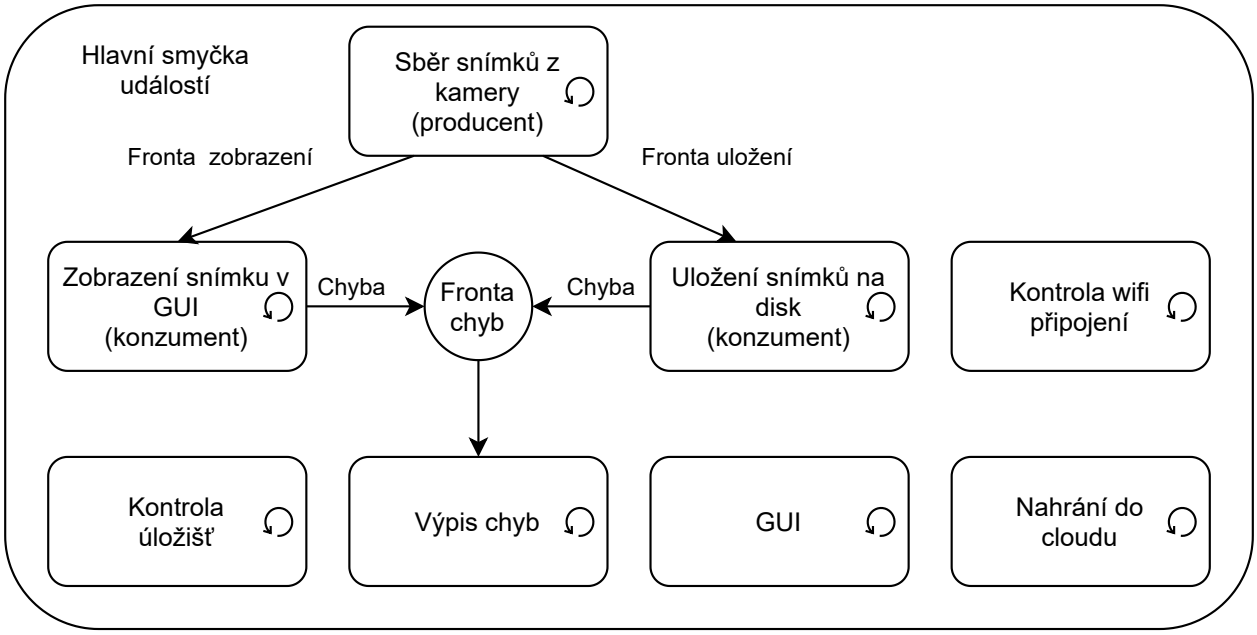
V praxi to, znamená, že jednotlivé funkce v aplikaci nejsou spouštěny jedna podruhé. Namísto toho je při spuštění aplikace vytvořeno několik na sobě nezávislých smyček (obrázek 4.3), kterým je přidělován výpočetní čas **hlavní smyčkou událostí**. Platí, že v jeden čas je v aplikaci vykonávána pouze **jedna instrukce** – nejedná se tedy o paralelismus.



Obrázek 4.1: Grafické uživatelské rozhraní aplikace kočárek určené pro sběr obrázků řepy. Hlavní okno.



Obrázek 4.2: Grafické uživatelské rozhraní aplikace kočárek určené pro sběr obrázků řepy. Vedlejší okno určené pro připojení na síť WiFi.



Obrázek 4.3: Diagram znázorňující hlavní smyčku událostí spolu s konkrétními smyčkami, které po přidělení výpočetního času vykonávají samotnou logiku aplikace. Jednotlivé smyčky jsou označeny **znakem smyčky**.

Pro **získání snímku** z čidla (kamery) a **uložení snímku** na disk byl použit návrhový vzor *producent-konzument*, při kterém mezi sebou dvě smyčky komunikují pomocí tzv. **fronty** – smyčka produkující data (snímky) tato data vkládá do fronty a smyčka zapisující data z fronty data vybírá. Producent zapisuje nejen do fronty k zápisu, ale také do fronty k **zobrazení** obrázku. Zobrazení snímku uživateli obstarává další smyčka (konzument). Toto je zřejmé z obrázku 4.3.

Ostatní smyčky v aplikaci jsou na sobě nezávislé. Konkrétně se jedná o smyčku pro kontrolu připojených úložišť, zobrazení chybových zpráv, běh grafického rozhraní, kontrolu připojení k internetu i synchronizaci pořízených dat do cloudové infrastruktury **AWS S3** po automatickém připojení k WiFi.

Výsledná aplikace kočárek byla používána doslova *v polních* podmínkách. Byly jí pořizovány snímky řepy v jednotlivých fázích růstu na různých polích i v různých podmínkách (plný sluneční svit, déšť). Proto byla pro nasazení aplikace pro sběr snímků vybrána platforma Raspberry pi 4 s operačním systémem **Linux Raspbian**. Tento operační systém je založen na linuxové distribuci Debian. K pořízení snímků byla použita průmyslová kamera **Basler**, konkrétně Basler acA1440-220uc. V prototypu samotného plecího stroje je plánováno použít stejnou kameru, ale algoritmus by měl být schopný dobré predikce i na snímku z jiné, neprůmyslové kamery (například z obyčejné webkamery). Kvalitní průmyslová kamera byla použita zejména proto, že aplikace vyžaduje tzv. **global shutter** kvůli tomu, že obraz je snímán při pohybu.

Při inicializaci kamery aplikace *kočárek* je nastaven režim upřednostňovaní nižší hodnoty zesílení

```
/usr/bin/ssh -i ${KEY} -N -R ${PUB_PORT}:localhost:22 ${PUB_USER}@${PUB_IP}
```

Výpis 4.1: Vytvoření reverzního tunelu pro umožnění vzdáleného přístupu na zařízení.

a zároveň je maximální doba expozice nastavena na hodnotu 500 , což je empiricky ověřená hodnota při které nedochází ke znatelnému rozmazání obrazu při sběru snímků za chůze (do 5 km/h). Prostor ve výhledu kamery je také osvětlen umělým osvětlením.

Vzhledem ke světové pandemii COVID-19 byla do aplikace implementována funkcionalita automatické zálohy nasbíraných dat (snímků) do cloudového úložiště **AWS S3 Bucket**. S tímto také souvisela implementace funkce pro připojení na WiFi síť i funkcionalita pro ověření dostupnosti internetového připojení jakožto zpětná vazba uživateli, který se snaží zařízení na WiFi připojit (obrázek 4.2).

Plně vzdálený přístup na přístroj na zařízení pro možné řešení produkčních chyb byl umožněn s využitím tzv. **reverzního tunelu** na server s veřejnou IP adresou. Díky tomuto řešení je umožněno vývojáři se na zařízení v produkci připojit (za podmínky, že je připojeno k internetu pomocí WiFi, nebo přímo kabelem) a řešit tak případné problémy v produkci. Pro představu je uveden příkaz pro vytvoření reverzního tunelu na výpisu 4.1. Po vytvoření reverzního tunelu je zařízení dostupné na serveru s veřejnou IP adresou PUB_IP na portu PUB_PORT, kde je vystaven port 22 produkčního zařízení pro sběr snímků. Přístup na zařízení je umožněn linuxovým nástrojem **ssh** [25].

Stav tohoto připojení je kontrolován, případně je vykonán pokus o vytvoření reverzního tunelu každou jednu minutu pomocí nástroje **crontab** [26]. V případě, že na zařízení není spuštěný žádný proces ssh, přístroj se pokusí vytvořit tunel. Počítá se s tím, že se to nemusí podařit. V takovém případě se zařízení pokusí vytvořit reverzní tunel za již zmíněnou minutu. Pokud ovšem na zařízení již nějaký ssh proces běží, pokus o připojení se neprovádí – předpokládá se, že tunel je již vytvořený. Pro správnou funkci je třeba dbát na to, aby na zařízení vývojář nikdy nespouštěl žádný jiný ssh proces než ten, který vytváří tunel – toto je ovšem okrajový případ a nepředpokládá se, že by to vývojář udělal.

4.2 Předzpracování datasetu

Pomocí kočárku bylo nasbíráno asi 60 000 snímků v rozlišení 1440x1080 (šířka x délka). V těchto snímcích byly poté s využitím nástroje **labelImg** [27] označeny jednotlivé rostliny cukrové řepy i se svou pozicí. Jako formát pro tyto značky, tzv. anotace snímků, byl použit standardní formát **VOC**. Každému označenému snímku byl tedy přiřazen soubor ve formátu XLM se stejným jménem. Příklad takovéto anotace je ve výpisu 4.2. Z výpisu je patrné, že samotný soubor nese mimo samotné informaci o anotaci také informaci o anotovaném snímku (cesta, název, rozměry). Tento formát je navíc lehce rozšiřitelný o anotace segmentace. Také umožňuje nastavit jednotlivým anotovaným

objektům různé obtížnosti, toho lze následně využít při trénování sítě. V této práci však byla obtížnost všech objektů stejná.

```
    <annotation>
    <folder>nazevSouboru</folder>
    <filename>nazevSnimkuKeKteremuNaleziAnotace</filename>
    <path>cestaKeSnimku</path>
    <size>
        <width>1080</width>
        <height>1440</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>sugar-beet</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>0</xmin>
            <ymin>1385</ymin>
            <xmax>23</xmax>
            <ymax>1439</ymax>
        </bndbox>
    </object>
</annotation>
```

Výpis 4.2: Příklad použitého formátu anotaci VOC.

Takto označené snímky byly následně rozstříhány na velikost 288x1080, aby poměr stran odpovídal poměru stran tzv. nebezpečné zóny. Podle toho, zda v rozstřiženém snímku byla anotace, nebo nebyla, tak tomuto snímku byla přidělena třída značící přítomnost, či nepřítomnost řepy. Celkem bylo takto získáno 95 000 snímků, z nichž polovina třídy **snímek s řepou** a polovina třídy **snímek bez řepy**. Proces rozstříhání snímků je znázorněn na obrázku 4.4. Na základě těchto rozstříhaných snímků byla síť naučena rozdělovat snímky o tomto poměru stran do dvou kategorií – snímek s řepou a snímek bez řepy.

Obrázek 4.4: Znázornění procesu rozstříhání získaných snímků z originálního rozlišení 1440×1080 na 288×1080.

Kapitola 5

Tvorba modelu

Model neuronové sítě představuje konkrétní algoritmus s konkrétními hodnotami (váhami) získanými v průběhu procesu trénování na základě poskytnutého trénovacího datasetu. Každý model je založen na tzv. architektuře. Architektura představuje abstraktní popis operací v modelu (bez konkrétně naučených hodnot trénovatelných parametrů). Nejprve je třeba zvolit a implementovat vhodnou architekturu, poté lze s použitím získaných dat natrénovat konkrétní model.

Hlavním požadavkem kladeným na architekturu, která by mohla být použita pro řešení úlohy, je vysoká rychlost vyhodnocení, což je blízce spojeno s nízkou výpočetní složitostí. Dalším požadavkem je, aby model nezabíral příliš mnoho místa v paměti (GPU mají omezenou paměť v jednotkách GB). Na základě těchto požadavků odpadají ty úplně nejpřesnější architektury sítí v ILSVRC, protože je hledán kompromis mezi přesností a HW náročností a tyto sítě se soustředí pouze na přesnost.

Jak bylo zmíněno v kapitole 2.2, přelomovou prací v oblasti vývoje architektur neuronových sítí vhodných pro nasazení na embedded zařízení či mobily je práce, která představila první model z rodiny MobileNet [20]. Modely z rodiny MobileNet namísto standardní operace konvoluce implementují tzv. hloubkově oddělitelnou operaci konvoluce (**depthwise separable convolution operation**), která snižuje výpočetní náročnost modelu až **devětkrát** v porovnání s použitím standardní konvoluce. Práce byla tedy zaměřena právě na druhou verzi této odlehčené architektury, **MobileNetV2**. Pro úplnost však bude nejprve popsána první verze architektury (MobileNet).

5.1 MobileNet

Velmi dobrého poměru mezi přesností a výpočetní náročností architektury MobileNet bylo dosaženo nahrazením standardní operace konvoluce hloubkově oddělitelnou konvolucí.

Tato má za úkol dvě věci, **filtrovat** a **třídit** vstupní tenzor. Hloubkově oddělitelná konvoluce tyto operace provádí ve dvou specializovaných vrstvách (fázích). Filtrace a kombinace je tedy provedena zvlášť. Tímto se stane velikost tenzoru konvolučních jader nezávislou na hloubce výstupního tenzoru.

Do standardní konvoluční vrstvy v neuronové síti vstupuje tenzor (tzv. *feature mapa*) F o rozměru $D_F \times D_F \times M$ a výstupem budiž tenzor (*feature mapa*) G o rozměru $D_F \times D_F \times N$, kde D_F je výška a šířka vstupní feature mapy (šířka a výška je přímo úměrná šířce a výšce obrázku). M je hloubka vstupního tenzoru a N je hloubka výstupního tenzoru. Standardní konvoluce je tedy parametrizována konvolučním jádrem K o rozměru $D_K \times D_K \times M \times N$, kde D_K je rozměr samotného konvolučního filtru (obvykle čtvercový), M a N představuje počet vstupních a výstupních kanálů (hloubka)[1]. Výstupní feature mapa konvoluční vrstvy je potom rovna

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m} \tag{5.1}$$

Pro úplnost je možno dodat, že koeficienty k, l, n představují iteraci přes rozměrové dimenze G, koeficienty i, j, m zase iteraci přes rozměrové dimenze K. Výpočetní složitost takové operace je potom

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{5.2}$$

Hloubkově oddělitelná konvoluce je oproti tomu provedena ve dvou oddělených vrstvách, a sice ve vrstvě hloubkové konvoluce a bodové konvoluce.

Hloubková konvoluce aplikuje jeden filtr na každý vstupní kanál tenzoru. Bodová konvoluce s filtrem 1×1 zajistí roztřídění vyfiltrovaných kanálů. Výstupem je lineární kombinace v hloubkové dimenzi tenzoru získaného hloubkovou konvolucí. Rovnice popisující vstup a výstup vrstvy hloubkové konvoluce je následující

$$\widehat{G}_{k,l,m} = \sum_{i,j} \widehat{K}_{i,j,m} \cdot F_{k+i-1,l+j-1,m} \tag{5.3}$$

Výpočetní složitost hloubkové konvoluce je poté

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F \tag{5.4}$$

Výpočetní složitost hloubkově oddělitelné konvoluce (která nahrazuje tradiční konvoluci) je součet složitosti hloubkové konvoluce a tradiční konvoluce při výšce a šířce filtru rovné jedné ($D_K = 1$ – bodová konvoluce)

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{5.5}$$

Faktor snížení výpočetní složitosti získáme podělením výpočetní složitosti hloubkové konvoluce výpočetní složitostí tradiční konvoluce

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2} \tag{5.6}$$

Architektura MobileNet pro konvoluci využívá filtr o velikosti $D_K \times D_K = 3 \times 3$, z čehož plyne

[1] Každému vstupnímu kanálu je přiřazena váha.

až devětkrát nižší počet operací násobení než v porovnání s implementací bez hloubkově oddělitelné konvoluce [20]. Právě počet operací násobení v algoritmu přispívá významnou měrou k jeho výpočetní složitosti a tedy i rychlosti.

MobileNet ale přeci jen využívá tradiční konvoluci a to na jednom jediném místě – v první konvoluční vrstvě která je zodpovědná za prohledávání vlastností ve vstupním obrázku. Po této vrstvě s tradiční konvolucí následují vrstvy výhradně s hloubkově oddělitelnou konvolucí. Po každé konvoluční vrstvě následuje vrstva batchnorm (normalizace batch) a nelinearita ReLU. Poslední, plně propojená vrstva je bez nelinearity a její výstup jde rovnou do vrstvy aplikující *softmax* pro klasifikaci [20].

5.2 MobileNetV2

V nové verzi architektury MobileNet, MobileNetV2 jsou tři významné prvky, které tuto architekturu činí jedinečnou z pohledu přesnosti predikce a její rychlosti. Jmenovitě to je již zmíněné nahrazení tradiční konvoluce konvolucí hloubkově oddělitelnou, lineární úzká hrdla (linear bottlenecks) a obrácený reziduál (inverted residual). Namísto pouhé hloubkově oddělitelné konvoluce, MobileNetV2 implementuje tzv. hloubkově oddělitelnou konvoluci s úzkým hrdlem a reziduálním spojením.

Myšlenka lineárních úzkých hrdel se odvíjí od faktu, že snímky které budou do sítě vstupovat (reálné snímky reálného světa), jsou kódovány v tzv. aktivačních tenzorech o rozměru $h_i \times w_i \times d_i$. Takto kódované snímky nezpůsobí, že by prostor v aktivačním tenzoru byl rovnoměrně využit po celém jeho definičním oboru. Jednoduše řečeno, nevyužije se plný potenciál daného tenzoru k přesnému zakódování informace. Místo toho se tvoří tzv. **manifold of interest**[2], což představuje podmnožinu všech možných tenzorů se stejnou velikostí. Předpokládá se, že informaci, která je nesena v těchto, řekněme řídkých, tenzorech, je možno zakódovat do hustšího tenzoru s menším rozměrem – informace je před předáním další vrstvě komprimována právě přes lineární úzké hrdlo. Tímto je neuronová síť nucena lépe zobecňovat při učení, snižuje se riziko přeučení a stoupá celková kvalita predikce sítě.

Obrácený reziduál se odvíjí od faktu, že v takto komprimovaném tenzoru, úzkém hrdle, by měly být veškeré významné informace z předchozí vrstvy. Při standardním reziduálním spojení nejsou ale spojeny komprimované vrstvy, nýbrž původní, nekomprimované aktivační tenzory. V architektuře MobileNetV2 jsou reziduální spojení provedena právě mezi komprimovanými tenzory. Reziduální spojení mají velký význam pro zlepšení procesu učení neuronové sítě – umožňují lepší propagaci gradientu při trénování, zamezují tzv. problému mizejícího gradientu. Obrácený reziduál je oproti klasickému ovšem paměťově mnohem úspornější.

[2] Informace se v tenzoru *shlukují* v určitých *oblastech*.

5.3 Implementace achitektury

Architektura implementovaná v této práci vychází z architektury MobileNetV2 [1]. V architektuře jsou tedy použity prvky jako obrácený reziduál, lineární úzká hrdla i hloubkově oddělená konvoluce. Pro vytvoření architektury použité v této práci byl použit nástroj *Keras*. Obecné shrnutí implementace je zachyceno v tabulkách 5.2 a 5.1. Následuje detailní popis architektury přímo jejím kódem, tento bude nejprve více abstraktní, poté budou předvedeny i jednotlivé funkce popisující detailní operace a vrstvy.

```
def MobileNetV2(input_shape,classes):
    inputs, x = input_stage(input_shape)
    x = bottleneck_stage(x, alpha=1)
    x = output_stage(x, last_block_filters=1280, classes=classes)

    model = training.Model(inputs, x, name='mobilenetv2')
    return model
```

Výpis 5.1: Definice funkce MobileNetV2. Tato funkce vrací instanci modelu.

Ve výpisu 5.1 je kód pro vytvoření instance modelu. Tento kód je velmi abstraktní (neznáme konkrétní implementaci funkcí), na druhou stranu je velmi přehledný a dává ucelenou představu o architektuře. Z kódu je zřejmé, že architekturu je možné rozdělit do tří fází – fáze vstupu (*input_stage*), fáze hrdel (*bottleneck_stage*) a fázi výstupu (*output_stage*). Dále budou popsány tyto tři fáze.

```
def input_stage(input_shape):
    input_image = tf.keras.layers.Input(shape=input_shape)
    x = tf.keras.layers.Conv2D(
        32, kernel_size=3, strides=(2, 2),
        padding='same', use_bias=False,
        name='Conv1'
        )(input_image)
    x = tf.keras.layers.BatchNormalization(
        axis=channel_axis, epsilon=1e-3,
        momentum=0.999, name='bn_Conv1')(x)
    return input_image, tf.keras.layers.ReLU(6., name='Conv1_relu')(x)
```

Výpis 5.2: Definice vstupní fáze modelu.

Fáze vstupu je vyobrazena ve výpisu 5.2. Možný rozměr vstupního snímku je pevně definován parametrem *input_shape*. Následuje jediná plně konvoluční vrstva *Conv2D* s 32 filtry. Tato vrstva má roven parametr *stride* 2×2, výška a šířka výstupního tenzoru je tedy dvakrát menší než je

výška a šířka vstupního obrázku. Na druhou stranu má výstupní tenzor mnohonásobně více kanálů (32) než vstupní obrázek (3 kanály RGB, nebo 1 kanál černobílý). Následuje vrstva pro normalizaci batch *BatchNormalization*. Tato vrstva přispívá k rychlejšímu trénování sítě i k výsledné kvalitě natrénované sítě (její schopnosti zobecňovat).

```
def bottleneck_stage(x, alpha):
    x = bottleneck(
        x, n_blocks=1, filters=16, first_stride=1,
        alpha=alpha, expansion=1, start_block_id=0)
    x = bottleneck(
        x, n_blocks=2, filters=24, first_stride=2,
        alpha=alpha, expansion=6, start_block_id=1)
    x = bottleneck(
        x, n_blocks=3, filters=32, first_stride=2,
        alpha=alpha, expansion=6, start_block_id=3)
    x = bottleneck(
        x, n_blocks=4, filters=64, first_stride=2,
        alpha=alpha, expansion=6, start_block_id=6)
    x = bottleneck(
        x, n_blocks=3, filters=96, first_stride=1,
        alpha=alpha, expansion=6, start_block_id=10)
    x = bottleneck(
        x, n_blocks=2, filters=160, first_stride=2,
        alpha=alpha, expansion=6, start_block_id=13)
    return bottleneck(
        x, n_blocks=1, filters=320, first_stride=1,
        alpha=alpha, expansion=6, start_block_id=16)
```

Výpis 5.3: Definice fáze úzkých hrdel modelu.

Ve výpisu 5.3 je vyobrazena fáze úzkých hrdel (*bottleneck_stage*). Z výpisu je patrné, že fáze se skládá celkem ze sedmi bloků operace úzkého hrdla a že jednotlivé bloky mají různý počet výstupních kanálů (*filters*), hodnotu *stride*, která říká kolikrát je výška a šířka výstupního tenzoru zmenšená oproti tenzoru vstupnímu. První blok má hodnotu expanze rovnu jedné, další bloky mají tuto hodnotu rovnu šesti. Výpis představuje implementaci tabulky 5.2, přičemž c v tabulce je rovno parametru *filters* (počet výstupních kanálů), n je rovno *n_blocks*, tedy počtu opakování hrdla, t je rovno *expansions* (význam je zřejmý z tabulky 5.1) a s je rovno *first_stride*. Výpis nedává hlubší informaci o tom, jak je operace *bottleneck* implementována, toto bude popsáno později.

Poslední, výstupní fáze modelu je vyobrazena ve výpisu 5.4. První vrstvou je zde konvoluční vrstva s velikostí konvolučního filtru 1×1, jedná se o tzv. **bodovou** konvoluci. Tato má 1280 kanálů

Tabulka 5.1: Popis jednotlivých vrstev v modulu *úzkého hrdla*. Neuronová síť v této práci se skládá ze 16 takovýchto hrdel [1].

Vstup	Operátor	Výstup
$h \times w \times k$	1×1 conv2d ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3×3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1×1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Tabulka 5.2: Shrnutí implementované architektury. Každý řádek popisuje jednu vrstvu sítě opakovanou n-krát. Všechny vrstvy dané sekvence (n) mají stejný počet výstupních kanálů c. První vrstva sekvence má hodnotu *stride* rovnu s, ostatní vrstvy v sekvenci mají hodnotu *stride* rovnu 1. Jádro všech konvolučních filtrů má rozměr 3×3. Expanzní faktor reprezentuje t, jeho vliv je popsán v tabulce 5.1 [1].

Vstup	Operátor	t	c	n	s
$480 \times 96 \times 3$	conv2d	-	32	1	2
$240 \times 48 \times 32$	hrdlo	1	16	1	1
$240 \times 48 \times 16$	hrdlo	6	24	2	2
$120 \times 24 \times 24$	hrdlo	6	32	3	2
$60 \times 12 \times 32$	hrdlo	6	64	4	2
$30 \times 6 \times 64$	hrdlo	6	96	3	1
$30 \times 6 \times 96$	hrdlo	6	160	3	2
$15 \times 3 \times 160$	hrdlo	6	320	1	1
$15 \times 3 \times 320$	conv2d 1×1	-	1280	1	1
$15 \times 3 \times 1280$	avgpool 15×3	-	-	1	-
$1 \times 1 \times 1280$	husté propojení	-	2	1	-

(parametr *last_block_filters*, patrné z 5.1). Po této vrstvě následuje vrstva normalizace batch a poté aktivační vrstva ReLU.

```
def output_stage(x, last_block_filters, classes):
    x = layers.Conv2D(
        last_block_filters, kernel_size=1,
        use_bias=False, name='Conv_1')(x)
    x = layers.BatchNormalization(
        axis=channel_axis, epsilon=1e-3,
        momentum=0.999, name='Conv_1_bn')(x)
    x = layers.ReLU(6., name='out_relu')(x)
    x = layers.GlobalAveragePooling2D()(x)
    return layers.Dense(classes, activation='softmax',
                        name='predictions')(x)
```

Výpis 5.4: Definice výstupní fáze modelu.

Kapitola 6

Výběr platformy pro praktické nasazení

Vyvíjený algoritmus je určen k rozpoznání řepy od plevele na poli. Je třeba počítat s tím, že možnosti výběru HW jsou omezené jednak typickým faktorem, cenou, ale také příkonem zařízení, které bude napájeno z traktoru. Vzhledem k tomu, že zpracování obrazu musí probíhat v takřka reálném čase, je vhodné zvolit HW s GPU značky NVIDIA, na kterém mohou běžet algoritmy napsané v knihovně TensorFlow až několikrát rychleji. Na základě těchto požadavků byly vytipovány 3 HW vývojové desky (viz. obrázek 6.1). Jedná se o vývojové desky s GPU značky Nvidia - Jetson Nano (100 $), NX (400 $) a AGX (1000 $). V tab. 6.1 jsou porovnány základní parametry zvolených HW platforem i výsledná rychlost na jednotlivých platformách. Cílem je najít platformu s optimálním poměrem ceny vůči výkonu.

Všechny vytipované HW platformy jsou vývojovými sadami, které nejsou určeny k průmyslovému použití. Tyto platformy nicméně obsahují tzv. System on Module – systém na modulu (SoM), který je možné zapojit na vlastní, průmyslově certifikovanou HW platformu, jež obsahuje veškeré potřebné periferie. Výhodou použití vývojové sady je fakt, že se jedná o hotové řešení, na kterém je možné principiálně ověřit funkčnost řešení.

Tato kapitola je úzce spjata s následující kapitolou 7, protože výběr HW se odvíjí od rychlosti právě implementovaného algoritmu. V kapitole 7 je zmíněno, že vybraný algoritmus je implementovaná neuronová síť MobileNetV2 s rozměrem vstupní feature mapy rovné $480 \times 96 \times 3$ (tedy RGB snímek). Rychlost tohoto algoritmu je možno vidět v tabulce 6.1. Z této tabulky je také zřejmé, že rychlost algoritmu na platformě Jetson Nano je plně dostačující (40 FPS - 25 ms). Na vyšší rychlosti již nemá význam cílit, hlavně kvůli limitaci mechanické části zařízení, které by mělo převzít informaci z vyvíjeného algoritmu o přítomnosti řepy. Z tohoto důvodu byla jako vhodná platforma pro nasazení zvolena právě platforma **Jetson Nano**. V následujících podkapitolách je pro přehled uveden podrobný popis původně zvažovaných HW platforem i vybrané platformy Jetson Nano.

Obrázek 6.1: HW vytipovaný pro možné nasazení algoritmu. Nvidia Jetson Nano (*vlevo*), Nvidia Jetson NX (*uprostřed*), Nvidia Jetson AGX (*vpravo*).

Tabulka 6.1: Porovnání parametrů a rychlosti inference architektury MobileNetV2 s rozměrem vstupu $480 \times 96 \times 3$ na vybraných zařízeních (Jetson {Nano, NX, AGX}).

Zařízení	TOPS	TFLOPS	Max. výkon (Watt)	{TOPS,TFLOPS} na Watt	Rychlost MobileNetV2 (FPS)
Jetson Nano	-	∼0,5	10	0.0472	40
Jetson NX	21	-	15	1,4	76
Jetson AGX	32	-	30	∼1	500

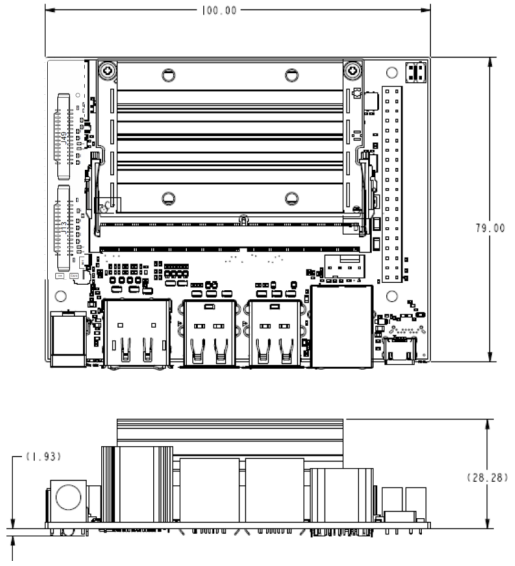
*Nelze přímo porovnávat TOPS a TFLOPS ani TOPS zařízení různého typu.

6.1 NVIDIA Jetson Nano

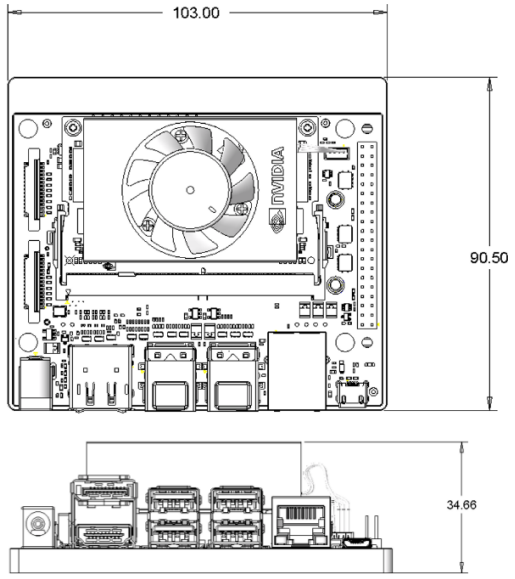
Vývojová sada NVIDIA Jetson Nano obsahuje SoM stejného jména. S cenou 100$ se jedná o nejlevnější sadu s NVIDIA Maxwell™ GPU se 128 CUDA jádry. Na rozdíl od ostatních produktů Jetson, grafická karta ve vývojové sadě Jetson Nano nepodporuje tenzorové operace (Tensor jádra). Proto není možné přímočaře porovnávat výpočetní výkon této grafické karty, který je v Tera Floating Point Operations Per Second – bilión operací s plovoucí čárkou za sekundu (TFLOPS) s výkonem ostatních zařízení uzpůsobených pro práci s tenzory třetího řádu, jejichž výkon je v Tensor Operations Per Second – tenzorové operace za sekundu (TOPS). Jedná se o jedinou vývojovou sadu, která má chlazení provedeno v podobě pasivního chladiče, což se dá považovat za výhodu. Sada dále obsahuje 64 bitový Quad-Core ARM® Cortex®-A57 MPCore CPU se čtyřmi jádry. Operační pamět v sadě je 64 bitová typu LPDDR4 s přenosovou rychlostí 25,6 GB/s a velikostí 4 GB.

6.2 NVIDIA Jetson NX

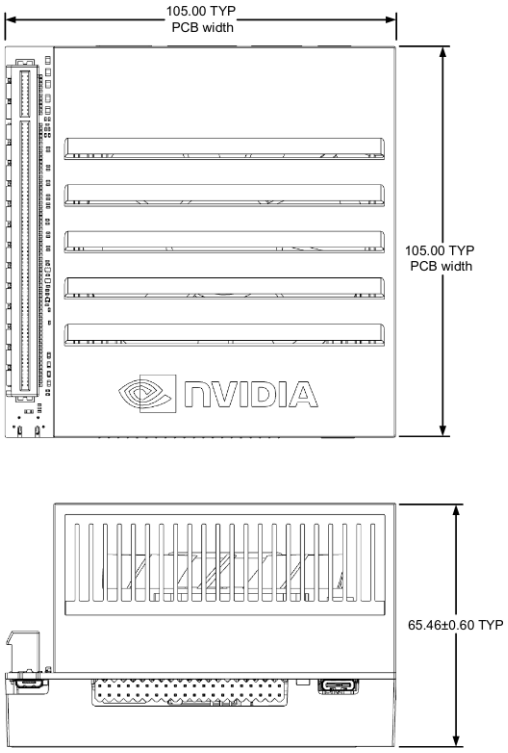
Vývojová sada NVIDIA Jetson NX stojí 400 $, je značně dražší než Jetson Nano. Obsahuje NVIDIA Volta™ GPU s 384 CUDA jádry, respektive 48 jádry CUDA Tensor. Díky jádrům CUDA Tensor je zvýšena propustnost GPU při počítání inference modelů. V sadě je dále 64 bitový NVIDIA Carmel ARM®v8.2 procesor, který umožňuje i vypnutí části jader při úsporném režimu. Operační paměť v sadě je 128 bitová typu LPDDR4x s přenosovou rychlostí 51,2 GB/s a velikostí 8 GB.



Obrázek 6.2: Mechanické schéma vývojové sady NVIDIA Jetson Nano.



Obrázek 6.3: Mechanické schéma vývojové sady NVIDIA Jetson NX.



Obrázek 6.4: Mechanické schéma vývojové sady NVIDIA Jetson AGX.

6.3 NVIDIA Jetson AGX

S cenou 1 000 $ je vývojová sada NVIDIA Jetson AGX nejdražším testovaným HW. V sadě je SoM s NVIDIA Volta™ GPU s 512 CUDA jádry, respektive 64 jádry CUDA Tensor. Srdcem sady je 64 bitový NVIDIA Carmel Arm®v8.2 procesor s osmi jádry umožňující vypnutí několik jader při úsporném režimu. Operační paměť v sadě je 256 bitová typu LPDDR4x s přenosovou rychlostí 136,5 GB/s a velikostí 32 GB.

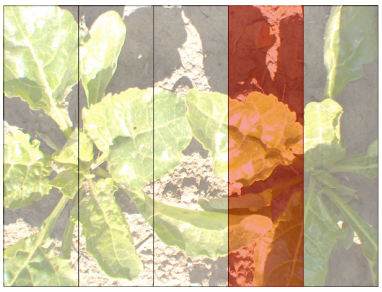
6.4 Jádro CUDA Tensor

Jako jádra CUDA Tensor jsou označovány nové výpočetní jednotky v dedikovaných grafických kartách NVIDIA. Tyto výpočetní jednotky jsou uzpůsobené právě pro vykonávání operace maticového násobení a přičtení další hodnoty k výsledku ($D = A \cdot B + C$). Jedná se tedy o CUDA jádra optimalizovaná pro vykonávání matematických operací nad tenzory.

Kapitola 7

Praktická implementace algoritmu

V této sekci bude podrobně popsána implementace architektury neuronové sítě pro řešení problematiky detekce řepy i proces trénování této architektury k vytvoření modelu. K implementaci byly použity nástroje z kapitoly 3, tedy jmenovitě programovací jazyk Python a knihovna pro strojové učení TensorFlow s nádstavbou Keras pro práci s neuronovými sítěmi. Cílem je vytvořit algoritmus, který podá binární informaci o přítomnosti řepy ve snímku. Pro tento problém použít algoritmus pro klasifikaci obrazu. Tento bude založen na architektuře MobileNetV2 [1]. Na obrázku 7.1 je znázorněn zvolený přístup pro získání informace, zda v nebezpečné zóně řepa je, nebo není. Ve snímku je vydefinována tzv. **nebezpečná oblast** a v této oblasti je řepa detekována. Název nebezpečná oblast byl zvolen proto, že se jedná o oblast, kde bude probíhat proces pletí plevele (tedy jeho ničení). V případě, že se v oblasti vyskytne řepa, proces pletí se na základě signálu získaného z algoritmu bude moci přerušit.



Obrázek 7.1: Vizualizace zvoleného přístupu pro detekci přítomnosti řepy v nebezpečné zóně. Červené pole znázorňuje nebezpečnou oblast, zde je obraz vyhodnocován. Obraz ze šedých polí je při inferenci zahozen.

```
{
    "runtimes": {
        "nvidia": {
            "path": "nvidia-container-runtime",
            "runtimeArgs": []
        }
    },
    "default-runtime": "nvidia"
}
```

Výpis 7.1: Konfigurace běhového prostředí docker pro zajištění přístupu ke grafické kartě značky NVIDIA. Obsah souboru /etc/docker/daemon.json

```
FROM nvcr.io/nvidia/l4t-tensorflow:r32.4.4-tf2.3-py3
RUN pip3 install click pandas
WORKDIR /work
ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8
CMD ["bash"]
```

Výpis 7.2: Specifikace pro sestavení docker obrazu pro prostředí měření rychlosti inference (Dockerfile).

7.1 Rychlost inference

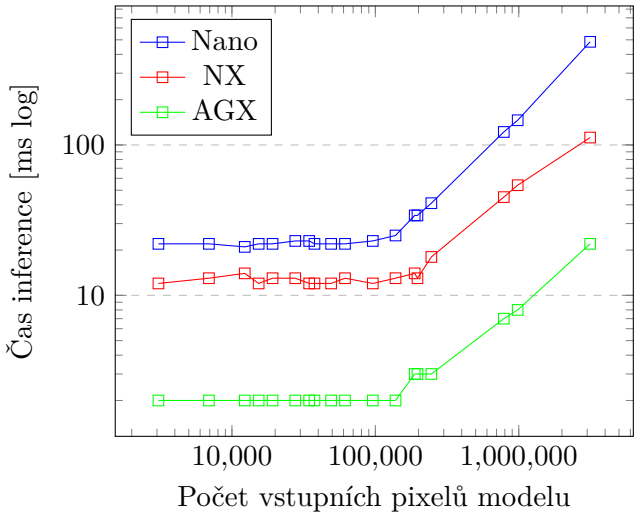
Protože se jedná o **realtime systém**, je rychlost algoritmu (tedy rychlost inference) stěžejní. Tato rychlost se odvíjí mimo jiné také od velikosti neuronové síťě, zejména od velikosti jejího vstupu. Proto byla rychlost inference změřena nejen na různých zařízeních, na které by bylo možné algoritmus nasadit, ale také při různé velikosti vstupu algoritmu.

Pro zaručení stejných podmínek (verze knihoven atd.) při měření na různých vytipovaných produktech z rodiny Jetson byla rychlost měřena v prostředí docker kontejneru, jehož specifikace pro sestavení docker obrazu je ve výpisu 7.2. Docker kontejner má minimální vliv na výkon a zároveň aplikaci určenou pro měření rychlosti inference umožňuje zabalit do vždy stejného prostředí knihoven. Před samotným měřením v docker kontejneru je třeba docker propojit s grafickou kartou NVIDIA, k tomu je třeba nainstalovat (konzolovou) aplikaci *nvidia-docker* a nakonfigurovat běhové prostředí dockeru tak, jak je popsáno ve výpisu 7.1. tímto způsobem bylo zajištěno stejné běhové prostředí na všech testovaných zařízeních.

Jak vyplývá z tabulky 7.1 a z obrázku 7.2, ve kterých je vyobrazena závislost počtu neuronů na době inference (čas potřebný pro získání predikce neuronové sítě), na jednotlivých zařízeních existuje určitá hranice rychlosti predikce, pod kterou není možné se dostat pouhým snížením počtu parametrů (velikosti vstupního obrázku). Na základě tohoto byla zvolena vstupní velikost (96 ×

480 × 3, kde 3 reprezentuje 3 RGB kanály), které odpovídá hodnota 138240 v tabulce 7.1 a na obrázku 7.2, tedy čas inference **25 ms** bez ztráty přesnosti (fp32). Toto vstupní rozlišení bylo zvoleno také proto, že pro výslednou funkčnost algoritmu není podstatná jen jeho rychlost, ale také jeho **přesnost**. Předpokládá se, že čím vyšší rozlišení vstupuje do neuronové sítě, tím více detailů je neuronová síť schopna rozlišit (jak je zřejmé z obrázku 7.3) a tím pádem lze předpokládat, že síť s vyšším vstupním rozlišením dosáhne také vyšší přesnosti predikce. Proto nebylo použito vstupní rozlišení nižší, s nevýznamně nižším časem inference (22 ms).

Z grafu na obrázku 7.2 je zřejmé, že rychlost inference je při zmenšovaní rozlišení vstupu od určitého bodu neměnná. To lze vysvětlit tím, že neuronová síť se skládá z několika desítek vrstev, přičemž každá vrstva potřebuje k výpočtu výstup z předchozí, na ní navazující vrstvy. **********Onen bod, od kterého rychlost inference se zmenšujícím se rozlišením vstupu je tedy pravděpodobně bodem, kdy už rychlosti vyhodnocení algoritmu nepomůže paralelizace výpočtu na grafické kartě právě z důvodu návaznosti těchto výpočtů.



Obrázek 7.2: Vykreslení závislosti velikosti vstupu (počet parametrů – pixelů) a času inference modelu MobileNetV2 pro různá zařízení (Nano, NX, AGX).

7.2 Trénování modelu

Tím úplně nejpodstatnějším parametrem, jenž podmiňuje vytvoření kvalitní neuronové sítě (schopné zobecnění), je použitý dataset. Pokud je dataset pro zvolenou úlohu příliš malý, často dochází k takzvanému přeučení sítě *overfitting* (vyobrazen na obrázku 7.5) – problémem je ale fakt, že neexistuje spolehlivá metoda, pomocí níž by bylo možné určit, zda je dostupný dataset pro danou úlohu dostatečný. Obecně ale platí, že čím větší je model neuronové sítě (čím více parametrů), tím větší má tato síť sklon k přeučení. Přesto lze přeučení sítě zabránit i v případě použití relativně malého datasetu pro trénování použitím tzv. **augmentace (zvětšení) dat**. **Augmentace dat** (v tomto

Tabulka 7.1: Tabulka závislosti velikosti rozlišení vstupu neuronové sítě na času inference modelu MobileNetV2 pro různá zařízení (Nano, NX, AGX).

		Čas inference (ms)		
Počet vstupních parametrů	Rozměr vstupu	Nano	NX	AGX
3072	$32 \times 32 \times 3$	22	12	2
6912	$48 \times 48 \times 3$	22	13	2
12288	$64 \times 64 \times 3$	21	14	2
19200	$80 \times 80 \times 3$	22	12	2
15360	$32 \times 160 \times 3$	22	12	2
27648	$96 \times 96 \times 3$	23	13	2
34560	$48 \times 240 \times 3$	23	12	2
37632	$112 \times 112 \times 3$	22	12	2
49152	$128 \times 128 \times 3$	22	12	2
61440	$64 \times 320 \times 3$	22	13	2
96000	$80 \times 400 \times 3$	23	12	2
138240	$96 \times 480 \times 3$	**25**	13	2
188160	$112 \times 560 \times 3$	34	14	3
196608	$256 \times 256 \times 3$	34	13	3
245760	$128 \times 640 \times 3$	41	**18**	3
786432	$512 \times 512 \times 3$	122	29	**7**
983040	$256 \times 1280 \times 3$	146	54	8
3145728	$1024 \times 1024 \times 3$	485	112	22

případě datasetu snímků) je proces, při kterém se nad každým datovým bodem (snímkem) provede tzv. třídu zachovávající transformace – tedy transformace, po které bude snímek stále identifikovatelný jako její původně určená třída (řepa identifikována, řepa neidentifikována). Příkladem takové transformace nechť je změna kontrastu, světlosti, saturace fotografie, nebo její otočení o x stupňů, či zrcadlové otočení, velmi užitečná je také změna kvality fotografie na základě **jpeg komprese** s různou kvalitou (vybrané příklady jsou na obrázku 7.4.

Trénink probíhal na stolním počítači s 16 jádrovým procesorem AMD Ryzen 9 5950X a hlavně nejmodernější grafickou kartou dostupnou pro stolní počítače – NVIDIA RTX 3090. Pro účel tréninku bylo nutné také správně nakonfigurovat systém Linux, zejména nainstalovat drivery pro grafickou kartu. Trénink jako takový probíhal v docker kontejneru, jehož definice (Dockerfile) je ve výpisu 7.3. Docker kontejner byl pro trénink použit zejména pro předejití možných konfliktů verzí různých knihoven (cuDNN atd.). Protože byla pro trénink použita nejnovější grafická karta NVIDIA, bylo nutné pracovat také s nejnovější verzí knihovny TensorFlow (tf-nightly).

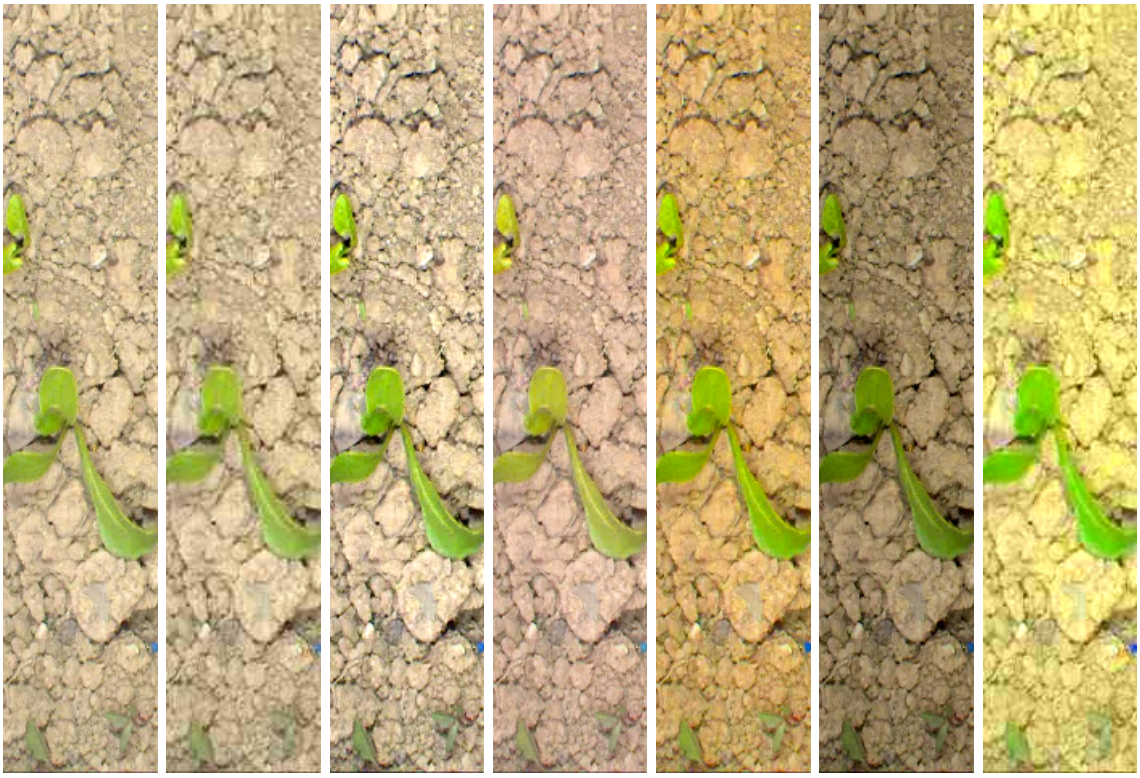
7.3 Výsledek trénování modelu

Pro trénování architektury byl použit celý dataset (92 000 snímků), rozdělený na dvě části (trénovací, testovací) v poměru $\frac{1}{5}$. Pro trénování bylo tedy použito 73 600 snímků a pro testování 18 400 snímků.

Obrázek 7.3: Příklady obrázků z datasetu vytvořeného pro tuto práci po rozstříhání. Vlevo nahoře – původní rozlišení 1080×288, vpravo nahoře – rozlišení zvolené jako nejvhodnější vzhledem k poměru rychlosti inference a zachování informace 480×96, dole – rozlišení, při kterém bylo dosáhnuto nejvyšší rychlosti inference 160 × 32 pixelů. Na každém obrázku je řepa. Nejméně patrná řepa je v každé sérii na obrázku vpravo – tato není při rozlišení 160 × 32 téměř detekovatelná ani člověkem. Při zvoleném rozlišení 480 × 96 pixelů je detekovatelná i pro člověka.

V průběhu jedné epochy jsou síti předvedeny všechny snímky z trénovacího i testovacího datasetu a na základě přesnosti předpovědi je vyhodnocena přesnost sítě. Snímky jsou v každé epoše lehce pozměněny tzv. **augmentací** a to tak, aby byly stále identifikovatelné jako daná třída. Jedná se o tzv. třídu zachovávající augmentaci. Augmentace se provádí proto, aby předpověď sítě byla dostatečně obecná a nedošlo k přeučení.

Z tabulky 7.2 a obrázku 7.6 je zřejmé, že přesnost predikce modelu na testovacím datasetu jen

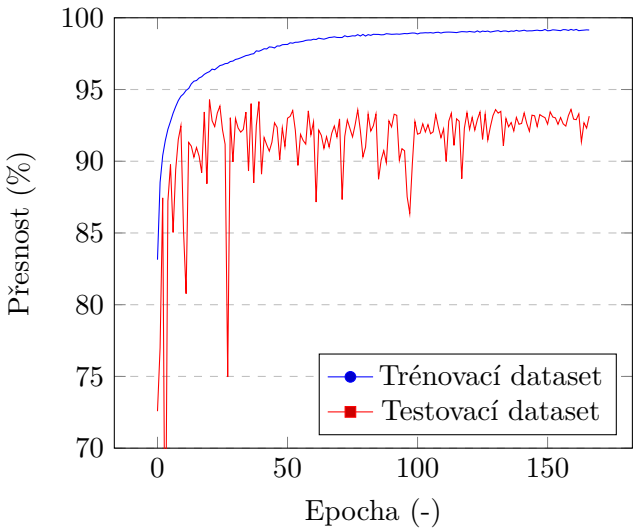


(a) (b) (c) (d) (e) (f) (g)

Obrázek 7.4: Příklady augmentovaných obrázků. Originální snímek[a]. Následují snímky, na které byly aplikovány následující augmentační transformace: JPEG komprese[b], změna kontrastu[c], HUE[d], saturace[e], světlosti[f]. Všechny augmentační transformace na jednom snímku[g].

```
FROM tensorflow/tensorflow:nightly-gpu

RUN pip install pascal-voc-tools==0.1.29 matplotlib click scipy sklearn tensorflow
    -addons pandas

RUN apt-get update -y && apt-get install libgl1-mesa-glx -y

WORKDIR /work

CMD ["bash"]
```

Výpis 7.3: Dockerfile pro kontejner, v němž probíhal trénink.



Obrázek 7.5: Příklad průběhu přesnosti na trénovacím a testovacím datasetu u přetrénovaného modelu.

výjimečně přesáhla přesnost na trénovacím datasetu v dané epoše. Zároveň není patrná tendence přesnosti na testovacím datasetu klesat se zvyšujícím se počtem epoch, naopak je přesnost na testovacím datasetu v kladné korelaci s přesností na trénovacím datasetu. Z toho lze vyvodit, že model trénovaný na augmentovaném datasetu nemá takovou tendenci k přeučení, je tedy schopný zobecnění. V praxi to znamená, že by model měl být schopný rozlišit řepu, kterou ještě neviděl a to i za různých světelných podmínek.

V praxi ovšem nejsou tato čísla tak podstatná jako faktické ověření funkčnosti a schopnosti algoritmu detekovat daný objekt za podmínek rozdílných od trénování i testování algoritmu. Byla tedy vybrána síť s nejvyšší přesností na testovacím datasetu a predikce této sítě byla vykreslena do videa, jehož snímky nebyly vůbec použity ani při testování, ani při trénování. Zhodnocením této predikce ve videu bylo dosaženo závěru, že síť je schopna velmi spolehlivě detekovat řepu v nebez-

pečné zóně na úkor výjimečné falešně pozitivní predikce. Původně bylo v plánu tuto funkcionalitu, kdy síť upřednostňuje falešně pozitivní predikci před falešně negativní, implementovat heuristicky (zvolením konstanty), toto nakonec není třeba. Vždy je vhodnější nevyplít plevel, než vyplít řepu. Toto lze vysvětlit tím, že v datasetu je mnoho fotek z chemicky ošetřovaného pole, kde mnoho plevele není.

Výsledkem trénování je poměrně přesný model (95 % na testovacím datasetu) natrénovaný za 1,5 dne na datasetu čítajícím 92 000 snímků řepy i bez řepy. Tento model byl také ověřen v praktickém nasazení, kdy jeho predikce byla vykreslena do videa a poté zhodnocena člověkem.

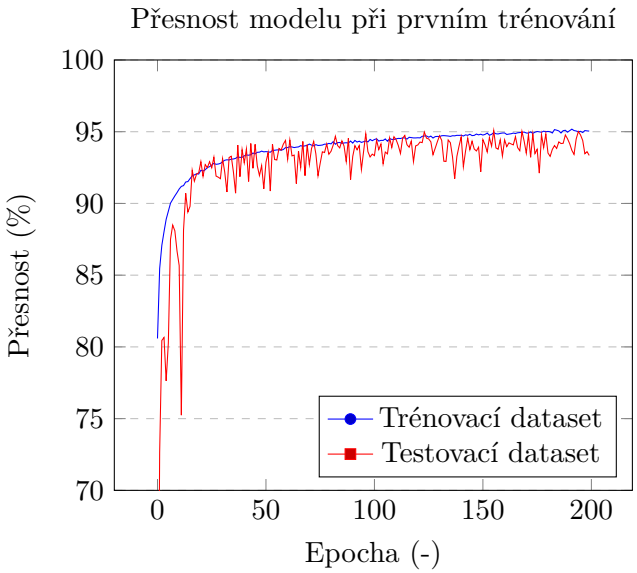
Tabulka 7.2: Zvolené hyperparametry pro první trénování sítě (vlevo). Výsledek trénování (vpravo).

Architektura	MobileNetV2
Počet epoch	200
Batch velikost	24
Snímků trénovacích	73 600
Snímků testovacích	18 400

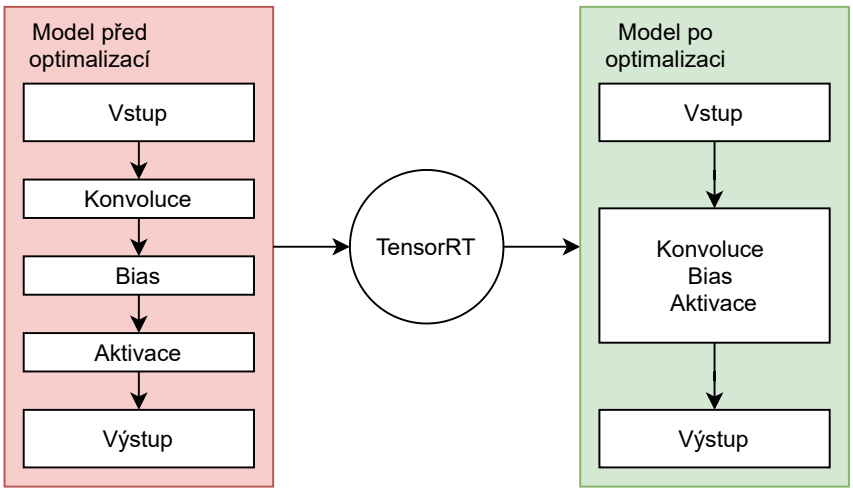
Epocha zvratu	50
Doba trénování	1,5 dne
Přesnost (trénovací)	95 %
Přesnost epocha 50 (test)	93 %
Přesnost epocha 200 (test)	93 %
Maximální přesnost (test)	95 % (epocha 123)

7.4 Optimalizace natrénovaného modelu

Po trénování s využitím knihoven TensorFlow a Keras je model s nejvyšší hodnotou přesnosti na testovacím datasetu uložen ve formátu nativním pro Keras. Model uložený ve formátu Keras je



Obrázek 7.6: Vykreslení průběhu přesnosti při prvním trénování. Přesnost je vyhodnocována na trénovacím i testovacím datasetu.



Obrázek 7.7: Znázornění optimalizačního mechanizmu TensorRT, konkrétně jde o fůzi vrstev.

jednak určen pro případné další trénování, ale je také přenositelný, může být tedy použit na jakékoli platformě. Toto je výhodné, pokud s modelem dynamicky pracujeme, zlepšujeme jeho přesnost trénováním, nebo jej chceme jednoduše spustit na různých platformách, případně jej dále zkoumat v Keras. V případě této práce je ale třeba model nasadit na určité zařízení, kde již nebude model trénován, ale bude na něm prováděna pouze inference – proces získání predikce. Pro tento účel je vhodné model optimalizovat. Vzhledem k tomu, že inference bude probíhat na zařízení s GPU značky NVIDIA, byl použit nástroj TensorRT. Tento nástroj upraví operace uložené v modelu tak, že inference proběhne jednak rychleji a jednak sníží nároky algoritmu na paměť zařízení. Toho je docíleno několika mechanismy, z nichž nejpodstatnější je mechanizmus fúze vrstev a tenzorů, který je znázorněn na obrázku 7.7. Díky mechanismu fúze vrstev a tenzorů se operace, u kterých je to možné, neprovádějí zvlášť, postupně, ale v jednom kroku.

Další možností, jak model optimalizovat, tedy snížit jeho paměťovou náročnost a zároveň zvýšit rychlost inference, je změna přesnosti z **fp32** na **fp16** případně i na **int8**[1]. Optimalizace modelu snížením jeho přesnosti ovšem nepřipadá v úvahu, proto v této práci nebyla použita. Navíc při optimalizaci na přesnost *int8* je jednak třeba kalibrace (snížení zobecnění modelu) a jednak operace s přesností *int8* nejsou podporovány na vybrané vývojové sadě Jetson Nano. Optimalizace byly tedy provedeny pouze na původní přesnost **fp32**.

[1]Zkratka fp znamená *floating point*, tedy datový typ *číslo s pohyblivou desetinnou čárkou*. Čísla 32 a 16 za touto zkratkou značí počet bitů které datový typ zabírá v paměti. Zkratka *int* znamená datový typ integer.

Kapitola 8

Závěr

Hlavním cílem této práce bylo popsat celý proces vývoje algoritmu strojového učení určeného pro úlohu zpracování obrazu. Konkrétně se jednalo o úlohu detekce zemědělské plodiny (cukrové řepy) ve snímku. Praktická část zahrnovala vytvoření datasetu, implementaci algoritmu neuronové sítě, natrénování implementované neuronové sítě na získaných datech (datasetu), výběr platformy pro nasazení a optimalizaci algoritmu pro nasazení.

Pro **vytvoření datasetu** byla v rámci této práce vyvinuta aplikace s pracovním názvem *kočárek*. Tato aplikace umožnila po nasazení na jednodeskový počítač Raspberry Pi 4 sběr snímků rostlin řepy pomocí průmyslové kamery Basler. Aplikace byla vyvinuta asynchronně a byla do ní implementována i funkcionalita pro připojení na WiFi síť a následné nahrání snímků do AWS S3 Bucketu (cloud). Více informací k této aplikaci je uvedeno v kapitole 4.1. Takto získaný dataset byl následně předzpracován způsobem popsaným v kapitole 4.2.

Implementace algoritmu pro zpracování snímků – neuronové sítě byla inspirována architekturou MobileNetV2 [1]. Tato architektura byla nejprve podrobně rozebrána v kapitolách 5.1 a 5.2. Poté byla popsána samotná implementace neuronové sítě v kapitole 7. Výsledný proces **trénování neuronové sítě** na získaných datech byl popsán v kapitole 7.3, samotným výsledkem je neuronová síť schopná predikovat přítomnost rostliny cukrové řepy na obrázku s přesností 95 %.

Jako vhodná **platforma pro nasazení** byl na základě informací v kapitole 6 vybrán jednodeskový počítač NVIDIA Jetson Nano. Toto zařízení v ceně 100 $ obsahuje dedikovanou grafickou kartu značky NVIDIA, kterou je možno využít pro akceleraci algoritmů strojového vidění implementovaných pomocí zvolené knihovny TensorFlow – více o této knihovně v kapitole 3.2. Tato konkrétní platforma byla zvolena, protože rychlost optimalizovaného algoritmu na ní je plně dostačující (40 snímků za sekundu). Navíc je její cena v porovnání s Jetson NX (400 $) a Jetson AGX (1 000 $) velmi přívětivá.

Proces **optimalizace neuronové sítě** je popsán v kapitole 7.3 a zároveň byl základem pro výběr optimální platformy pro nasazení, protože testování rychlosti algoritmu na zmíněných platformách z rodiny NVIDIA Jetson bylo prováděno po optimalizaci tohoto algoritmu pro jednotlivé platformy.

Pro optimalizaci byla použita knihovna TensorRT, která díky fúzi vrstev a tenzorů umožňuje až několikanásobné zrychlení inference algoritmu (proces získání predikce). Více k procesu optimalizace algoritmu je uvedeno v kapitole 7.4.

Z pohledu dalšího vývoje by mohl být algoritmus vytvořený v rámci této diplomové práce použit v mechanickém systému pro automatické odstraňování plevele v poli cukrové řepy. Praktickým nasazením optimalizovaného algoritmu na zařízení NVIDIA Jetson Nano bylo ověřeno, že je s jeho pomocí možné zpracovávat vstupní snímky s rychlostí až 25 ms (40 snímků za sekundu) a to s přesností rozpoznání 95 %.

Bibliografie

1. SANDLER, Mark; HOWARD, Andrew; ZHU, Menglong; ZHMOGINOV, Andrey; CHEN, Liang-Chieh. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. 2019. Dostupné z arXiv: 1801.04381 [cs.CV].
2. *Zdrojový kód Linuxového jádra*. [N.d.]. Dostupné také z: https://github.com/torvalds/linux. Navštíveno: 31.12.2020.
3. ABADI, Martín; AGARWAL, Ashish; BARHAM, Paul; BREVDO, Eugene; CHEN, Zhifeng; CITRO, Craig; CORRADO, Greg S.; DAVIS, Andy; DEAN, Jeffrey; DEVIN, Matthieu; GHEMAWAT, Sanjay; GOODFELLOW, Ian; HARP, Andrew; IRVING, Geoffrey; ISARD, Michael; JIA, Yangqing; JOZEFOWICZ, Rafal; KAISER, Lukasz; KUDLUR, Manjunath; LEVENBERG, Josh; MANÉ, Dandelion; MONGA, Rajat; MOORE, Sherry; MURRAY, Derek; OLAH, Chris; SCHUSTER, Mike; SHLENS, Jonathon; STEINER, Benoit; SUTSKEVER, Ilya; TALWAR, Kunal; TUCKER, Paul; VANHOUCKE, Vincent; VASUDEVAN, Vijay; VIÉGAS, Fernanda; VINYALS, Oriol; WARDEN, Pete; WATTENBERG, Martin; WICKE, Martin; YU, Yuan; ZHENG, Xiaoqiang. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Dostupné také z: https://www.tensorflow.org/.
4. *Robovátor*. [N.d.]. Dostupné také z: https://www.robovator.com/.
5. HUBEL, D. H.; WIESEL, T. N. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*. 1959-10, roč. 148, č. 3, s. 574–591. ISSN 0022-3751. Dostupné z DOI: 10.1113/jphysiol.1959.sp006308. PMC1363130[pmcid].
6. MARR, David. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. USA: Henry Holt a Co., Inc., 1982. ISBN 0716715678.
7. KUNIHIKO, Fukushima; SEI, Miyake. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition. In: AMARI, Shun-ichi; ARBIB, Michael A. (ed.). *Competition and Cooperation in Neural Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, s. 267–285. ISBN 978-3-642-46466-9.

8. LECUN, Y.; BOSER, B.; DENKER, J. S.; HENDERSON, D.; HOWARD, R. E.; HUBBARD, W.; JACKEL, L. D. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*. 1989, roč. 1, č. 4, s. 541–551. Dostupné z DOI: 10.1162/neco.1989.1.4.541.

9. VIOLA, Paul; JONES, Michael. Rapid Object Detection using a Boosted Cascade of Simple Features. In: 2001-02, sv. 1, s. I–511. ISBN 0-7695-1272-0. Dostupné z DOI: 10.1109/CVPR.2001.990517.

10. *Web PASCAL VOC*. [N.d.]. Dostupné také z: http://host.robots.ox.ac.uk/pascal/VOC/.

11. EVERINGHAM, Mark; GOOL, Luc; WILLIAMS, Christopher K.; WINN, John; ZISSERMAN, Andrew. The Pascal Visual Object Classes (VOC) Challenge. *Int. J. Comput. Vision*. 2010-06, roč. 88, č. 2, s. 303–338. ISSN 0920-5691. Dostupné z DOI: 10.1007/s11263-009-0275-4.

12. *Web ImageNet*. [N.d.]. Dostupné také z: http://www.image-net.org/.

13. RUSSAKOVSKY, Olga; DENG, Jia; SU, Hao; KRAUSE, Jonathan; SATHEESH, Sanjeev; MA, Sean; HUANG, Zhiheng; KARPATHY, Andrej; KHOSLA, Aditya; BERNSTEIN, Michael; BERG, Alexander C.; FEI-FEI, Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*. 2015, roč. 115, č. 3, s. 211–252. Dostupné z DOI: 10.1007/s11263-015-0816-y.

14. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. Lake Tahoe, Nevada: Curran Associates Inc., 2012, s. 1097–1105. NIPS'12.

15. *ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)*. [N.d.]. Dostupné také z: http://image-net.org/challenges/LSVRC/2012/results.html.

16. SIMONYAN, Karen; ZISSERMAN, Andrew. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. Dostupné z arXiv: 1409.1556 [cs.CV].

17. SZEGEDY, C.; WEI LIU; YANGQING JIA; SERMANET, P.; REED, S.; ANGUELOV, D.; ERHAN, D.; VANHOUCKE, V.; RABINOVICH, A. Going deeper with convolutions. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, s. 1–9. Dostupné z DOI: 10.1109/CVPR.2015.7298594.

18. MA, Ningning; ZHANG, Xiangyu; ZHENG, Hai-Tao; SUN, Jian. *ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design*. 2018. Dostupné z arXiv: 1807.11164 [cs.CV].

19. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep Residual Learning for Image Recognition. *CoRR*. 2015, roč. abs/1512.03385. Dostupné z arXiv: 1512.03385.

20. HOWARD, Andrew G.; ZHU, Menglong; CHEN, Bo; KALENICHENKO, Dmitry; WANG, Weijun; WEYAND, Tobias; ANDREETTO, Marco; ADAM, Hartwig. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*. 2017, roč. abs/1704.04861. Dostupné z arXiv: 1704.04861.

21. HOWARD, Andrew; SANDLER, Mark; CHU, Grace; CHEN, Liang-Chieh; CHEN, Bo; TAN, Mingxing; WANG, Weijun; ZHU, Yukun; PANG, Ruoming; VASUDEVAN, Vijay; LE, Quoc V.; ADAM, Hartwig. Searching for MobileNetV3. *CoRR*. 2019, roč. abs/1905.02244. Dostupné z arXiv: 1905.02244.

22. IOFFE, Sergey; SZEGEDY, Christian. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. Dostupné z arXiv: 1502.03167 [cs.LG].

23. FOUNDATION, Python Software. *Dokumentace jazyka Python verze 3.9*. [N.d.]. Dostupné také z: https://docs.python.org/3/. Navštíveno: 31.12.2020.

24. *Oficiální dokumentace knihovny TensorRT – podporované operace*. [N.d.]. Dostupné také z: https://docs.nvidia.com/deeplearning/tensorrt/support-matrix/index.html.

25. *Dokumentace linuxového nástroje SSH*. [N.d.]. Dostupné také z: https://man.openbsd.org/ssh.

26. *Dokumentace linuxového nástroje crontab*. [N.d.]. Dostupné také z: https://man7.org/linux/man-pages/man5/crontab.5.html.

27. *Oficiální stránka nástroje pro tvorbu anotací labelImg*. [N.d.]. Dostupné také z: https://github.com/tzutalin/labelImg.